Softwareprojekt

GalFld

Eine Umsetzung endlicher Körper in der Programmiersprache Haskell mit besonderer Betrachtung der Faktorisierung von Polynomen über endlichen Körpern

vorgelegt von

Stefan Hackenberg und Maximilian Huber

am

Institut für Mathematik
der
Universität Augsburg

betreut durch

Prof. Dr. Dirk Hachenberger

abgegeben am

09.09.2014

Inhaltsverzeichnis

1 Haskell

http://xkcd.com/1312/

1.1 Über die Programmiersprache

Lipovača beschreibt in der Einleitung zu [11] Haskell wie folgt:

Haskell ist eine *rein funktionale* Programmiersprache. Im Gegensatz zu *imperativen* Programmiersprachen, bei denen man dem Computer eine Folge von ausführbaren Aufgaben übergibt (Strukturen, die diesen Ablauf steuern, wären beispielsweise `for` und `while`), bestehen Programme in rein funktionalen Sprachen aus einer Menge von Funktionsdefinitionen, die man als Abbildungen von Eingabedaten auf Ausgabedaten verstehen kann.

Möchte man beispielsweise die Fakultät einer Zahl berechnen, so gibt man in einer imperativen Sprache die konkret notwendigen Anweisungen, um die Fakultät eines Eingabedatums zu berechnen; in einer rein funktionalen Sprache definiert man dagegen die

Fakultät als rekursive Abbildung folgendermaßen:

$$\begin{aligned} \_! : \quad \mathbb{N} &\to \mathbb{N} \\ n &\mapsto \begin{cases} n \cdot (n-1)!, & n > 0\,, \\ 1, & n = 0\,. \end{cases} \end{aligned}$$

In rein funktionalen Programmiersprachen sind die Werte von Variablen unveränderbar (sog. *immutable objects*) und Funktionen haben keine *Wirkungen* (im Sinne der Informatik (engl. *side-effects*)). Betrachtet man den Computer als abstrakte Turing-Maschine, so ändert eine Funktion einer rein funktionalen Programmiersprache den Zustand der Maschine nicht. Funktionen können ausschließlich auf ihren Eingaben basierende Berechnungen durchführen. Dies scheint eine Einschränkung zu sein, hat aber in der Tat einige Vorteile. Beispielsweise liefert eine Funktion bei gleichen Eingaben, unabhängig von der Umgebung, immer den gleichen Rückgabewert (sog. *referentielle Transparenz*).

Haskell ist *lazy*. Das bedeutet, dass Funktionen nicht ausgewertet werden, solange das Ergebnis nicht benötigt wird. Dies wird durch referentielle Transparenz ermöglicht. Haskell bemüht sich, die Auswertung von Ausdrücken so lange wie möglich zu vermeiden. Dadurch können auch unendliche Datenstrukturen verwendet werden, solange sie irgendwann auf einen endlichen Teil reduziert werden. `take 5 [1..]` liefert beispielsweise die ersten 5 Elemente der scheinbar unendlichen Liste der natürlichen Zahlen beginnend bei 1.

Haskell ist *statisch typisiert*. Das bedeutet, dass der Computer bereits zur Compilezeit zwischen Typen unterscheidet. Auf diese Weise können viele Fehler bereits vor dem eigentlichen Ausführen des Programms erkannt werden.

Darüber hinaus kann Haskell Typen *inferieren*, d.h. die Angabe eines Typs ist meist nicht zwingend erforderlich. So hat zum Beispiel `[1::Int, 2, 3]` die gleiche Bedeutung wie `[1::Int, 2::Int, 3::Int]`.

Die Geschichte von Haskell begann 1987, als „some really smart guys [...] got together to design a kick-ass language.“ [11, Section 1] Der *Haskell Report*, welcher die erste stabile Version beschreibt, wurde 1999 publiziert (überarbeitete Version: [14]). Der aktuelle Standard wird beschrieben in [12].

Gute Bücher zum Einstieg in Haskell sind [10] und [11]. Eine ausführlichere Liste findet sich unter `http://www.haskell.org/haskellwiki/Books` und einige Tutorials bietet `http://www.haskell.org/haskellwiki/Tutorials`.

1.2 Ausführen von Haskell-Programmen

Haskell kann jederzeit interpretiert oder compiliert werden. Mit dem Interpreter `ghci` (Glasgow Haskell Compiler Interpreter) oder `hugs` kann man einfach Programme oder Programmabschnitte testen. Alternativ erhält man durch Compilieren mit `ghc` ausführbare Dateien, welche dank umfangreicher Optimierung performanter sind. Für eine ausführlichere Optimierung gibt es den Compiler-Parameter `-O`. Ein noch besseres Ergebnis verspricht `-O2` auf Kosten der Dauer des Compilierens.

Die Compiler-Option `-threaded` bereitet die ausführbare Datei darauf vor, parallel ausgeführt zu werden. Wird diese mit `-RTS -N`*X* gestartet, so nutzt das Programm *X* Prozessorkerne.

1.3 Installieren von Haskell-Paketen

Haskell-Pakete installieren sich am besten mit dem Konsolenwerkzeug `cabal`[1] für Windows und Unix-Systeme.

```
cabal update
```

liest die aktuell verfügbare Paketliste von `https://hackage.haskell.org` und

```
cabal install PAKETNAME
```

installiert ein Paket aus jener Bibliothek. Dabei löst `cabal` selbstständig notwendige Abhängigkeiten auf.

1.4 Entwicklung von Haskell-Code

Für Haskell gibt es eine umfangreiche Auswahl an Programmen, die bei der Entwicklung von Haskell-Bibliotheken und -Programmen helfen. Die Folgenden wurden für dieses Projekt genutzt.

1.4.1 Tests: `hspec`

> *Hspec is roughly based on the Ruby library RSpec. However, Hspec is just a framework for running HUnit and QuickCheck tests. Compared to other options, it provides a much nicer syntax that makes tests very easy to read.*[2]

[1] `http://www.haskell.org/cabal/download.html`
[2] `https://hackage.haskell.org/package/hspec`

`Hspec` ermöglicht es, Funktionen auf verschiedenste Eingabeparameter zu testen.

Ein einfaches und selbsterklärendes Beispiel[3] ist

```
-- Datei Spec.hs
import Test.Hspec
import Test.QuickCheck
import Control.Exception (evaluate)

main :: IO ()
main = hspec $ do
  describe "Prelude.head" $ do
    it "returns the first element of a list" $ do
      head [23 ..] `shouldBe` (23 :: Int)

    it "returns the first element of an *arbitrary* list" $
      property $ λx xs → head (x:xs) ≡ (x :: Int)

    it "throws an exception if used with an empty list" $ do
      evaluate (head []) `shouldThrow` anyException
```

Ein Ausführen, beispielsweise durch `runhaskell Spec.hs`, liefert die folgende Konsolenausgabe:

```
Prelude.head
  - returns the first element of a list
  - returns the first element of an *arbitrary* list
  - throws an exception if used with an empty list

Finished in 0.0028 seconds
3 examples, 0 failures
```

1.4.2 Benchmarking: `criterion`

> *This library provides a powerful but simple way to measure software performance. It provides both a framework for executing and analysing benchmarks and a set of driver functions that makes it easy to build and run benchmarks, and to analyse their results.*[4]

Um Funktionen mit sehr kurzer Ausführungszeit zu vergleichen oder um statistische Schwankungen auszugleichen, werden Tests mehrfach ausgeführt.

Damit das Ergebnis trotz Lazyness vollständig ausgewertet wird, muss der Typ der Berechnung eine Instanz von `NFData`.

Weitere Frameworks für Tests sind beispielsweise

[3] `http://hspec.github.io/`

[4] `https://hackage.haskell.org/package/criterion`

- `QuickCheck`,
- `SmallCheck` und
- `QuickSpec`.

Darüber hinaus gibt es das Paket `Tasty`, welches mehrere Frameworks unter einer einheitlichen API zusammenfasst.

1.4.3 Zusammenfügen: `cabal`

> *The Haskell Common Architecture for Building Applications and Libraries: a framework defining a common interface for authors to more easily build their Haskell applications in a portable way.*
>
> *The Haskell Cabal is part of a larger infrastructure for distributing, organizing, and cataloging Haskell libraries and tools.*[5]

`cabal` ist ein Paketmanager, kann aber auch als Build-System benutzt werden. Die Konfiguration übernimmt eine `.cabal` Datei, welche das Paket und seinen Inhalt definiert (siehe z.B. `galfld.cabal`). Auch kann darin definiert werden, wo sich Tests oder Benchmarks befinden, damit diese zentral ausgeführt werden können.

Ein relativ neues Feature (ab Versionen >1.18) sind *Sandboxen*. Diese erzeugen eine virtuelle Umgebung, in der benötigte Pakete lediglich lokal installiert werden. Die systemweite Konfiguration von `cabal` bleibt davon unberührt.

Gute Anleitungen finden sich unter HaskellWiki[6].

1.4.4 Dokumentation: `haddock`

> *Haddock is a tool for automatically generating documentation from annotated Haskell source code.*[7]

Das Programm `haddock` nutzt die Kommentare im Quellcode um daraus eine standardisierte HTML-Dokumentation zu erstellen.

[5] `https://hackage.haskell.org/package/Cabal`
[6] `http://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program`
[7] `http://www.haskell.org/haddock/`

1.5 Das Haskell-Typensystem

Da in Haskell bereits zur Compilezeit klar ist, welchen Typs jeder Ausdruck ist, lassen sich Fehler, wie das Dividieren eines `Bool` durch einen `Int`, bereits vor dem Ausführen entdecken. Gekennzeichnet werden Typenangaben durch :: als Infix-Operator.

```
count :: Int
```

beispielsweise erklärt die Variable `count` zum Typen `Int`. Da es in Haskell Funktionen höherer Ordnung (engl. *higher-order-functions*) gibt, ist eine Typenangabe bei Funktionen obligatorisch. So hat die Funktion `head`, welche das erste Element einer List wiedergibt, den Typ:

```
head :: [a] → a
```

Zu bemerken ist hier, dass für obiges Beispiel der Typ der Elemente der Liste nicht festgelegt ist, d.h. der Platzhalter `a` steht für jeden Typen. Ergo liefert `head` angewandt auf eine Variable vom Typ `[String]` einen Wert des Typs `String`, auf `[Int]` einen `Int`, etc.

Weiter gibt es Typen-Klassen, welche beispielsweise `interfaces` in Java ähneln. Eine Typ-Klasse beschreibt Funktionen und Eigenschaften, die dem Typ zu Eigen sind. Ein gutes Beispiel hierfür ist die Typ-Klasse `Eq`, welche als einzige Funktion den Operator `(≡)`[8] enthält und definiert[9] ist durch

```
class  Eq a  where
  (≡), (≠) :: a → a → Bool

      -- Minimal complete definition:
      --        (==) or (/=)
  x ≠ y     =  not (x ≡ y)
  x ≡ y     =  not (x ≠ y)
```

Möchte man nun die Gleichheit auf Listen als Gleichheit der Elemente implementieren, so könnte man `[a]` eine `instance` von `Eq` geben:

```
instance (Eq a) ⇒ [a] where
  xs ≡ ys  = and (zipWith (≡) xs ys)
```

Natürlich muss man nun – anders als in der Definition der Typ-Klasse – `(≡)` mit konkreter Bedeutung füllen. ⇒ beschreibt dabei eine Typen-Klassen-Restriktion. Hier darf also die Typen-Variable `a` nicht durch alle Typen ersetzt werden, sondern nur durch die, die eine Instanz `Eq` haben.

Die Typen-Klasse `Eq` stellt zunächst die beiden Prädikate `≡` und `≠` bereit. Da sich die eine Funktion aber jeweils durch Negation der anderen ergibt, wie in den letzten beiden

[8] Die Schreibweise mit Klammern notiert Infix-Operatoren.
[9] http://www.haskell.org/onlinereport/standard-prelude.html

Zeilen der Klassendefinition zu lesen ist, reicht es, lediglich eine der beiden Varianten für eine vollständige Definition anzugeben.

Eine umfangreiche Liste an wichtigen Typ-Klassen sowie eine ausführlichere Erklärung des Typensystems findet man z.B. im zweiten Kapitel von [11].

1.6 Pragmas

Pragmas[10] bieten in Haskell die Möglichkeit, für den Compiler bestimmte Kommandos in den Quellcode zu integrieren. Diese beeinflussen nicht die Bedeutung des Codes, sondern haben eher Einfluss auf die Effizienz des generierten Programms. Auch Haskell-Erweiterungen können damit aktiviert werden.

Ein (Sprach-)Pragma im Code ist berandet durch `{-#... #-}`, wie z.B. in

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE TemplateHaskell #-}
```

Dadurch werden die (Sprach-)Erweiterungen `CPP` und `TemplateHaskell` aktiviert. Die erste Erweiterung ermöglicht durch die Befehle `#if 1` bzw. `#if 0`, `#else` und `#endif` – analog zu `\iftrue` und `\iffalse` in LaTeX – mehrere Zeilen im Quellcode zu (de-)aktivieren. `TemplateHaskell` ermöglicht es, zusammen mit `QuasiQuotes` Haskell-Funktionen bereits zur Compilezeit auszuführen. Damit lässt sich auf dynamische Weise Code erzeugen oder auch Rechenaufgaben in die Compilezeit verlagern.

Darüber hinaus wollen wir folgende weiteren Pragmas vorstellen:

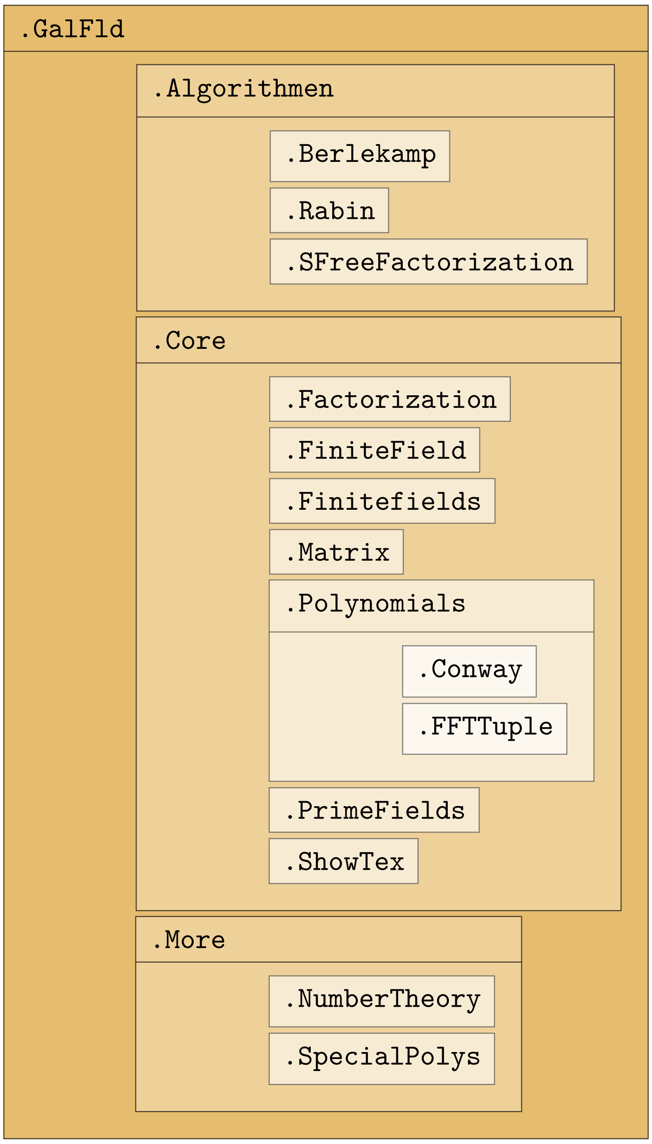
- `OPTIONSλ_GHC` Pragmas bieten eine Möglichkeit, dem Compiler Parameter spezifisch für die aktuelle Datei zu übergeben.
- `INLINE`-Pragmas werden in der Form `{-#INLINE funktionsname #-}` einer Funktion direkt vorangestellt und und geben dem Compiler die Anweisung, den Inhalt der Funktion anstelle des Funktionsaufrufs bei einer Benutzung zu setzen. Wird innerhalb eines Programms eine Funktion, nennen wir sie `foo`, benutzt, so setzt der Compiler an diese Stelle lediglich eine Referenz auf die Funktion `foo`, deren eigentliche Definition an irgendeiner anderen Stelle gespeichert wird. Das `INLINE`-Pragma fordert nun den Compiler auf, die gesamte Definition von `foo` statt einer Referenz zu setzen. Dies spart bei der Ausführung – gerade wenn die Funktion häufig mit wechselnden Argumenten aufgerufen wird, wie beispielsweise eine Addition – Zeit, da das Programm die aktuelle Position der Ausführung nicht verlassen muss. Dies geht jedoch auf Kosten einer gewissen Lazyness: Nehmen wir an, `foo` hätte die Deklaration `foo ::a →a` und wir würden in einem fiktiven Programm sehr oft `foo` x mit dem selben Argument x aufrufen, so würde ohne `INLINE` Haskell lediglich *einmal* `foo` x berechnen und die anderen Aufrufe durch das Ergebnis ersetzen. Durch

[10] https://www.haskell.org/ghc/docs/7.0.4/html/users_guide/pragmas.html

`INLINE` wird `foo` sofort durch seinen Inhalt ersetzt, wodurch *alle* `foo` x separat berechnet werden.

2 Implementierung

Struktureller Aufbau des Projektes.



2.1 Implementierung von Polynomen

`GalFld/Core/Polynomials.hs`
`GalFld/Core/Polynomials/FFTTuple.hs`
`GalFld/Core/Polynomials/Conway.hs`

2.1.1 Der Datentyp

Grundsätzlich gibt es zwei verschiedene Möglichkeiten Polynome zu implementieren: *sparse* und *dense*, d.h.

- entweder entscheidet man sich, ein Polynom $f(X) = a_n X^n + \ldots + a_0$ als Liste[1] der Länge $n+1$ zu hinterlegen,
- oder man speichert lediglich eine Liste von Tupeln (i, a_i), sodass $i \in \{0, \ldots, n\}$ den Index/Exponenten des Koeffizienten a_i angibt und alle Koeffizienten, die Null sind, ausgelassen werden.

In der hier vorliegenden finalen Implementierung haben wir uns für letztere Variante entschieden, da diese insbesondere bei spärlich besetzten Polynomen mit hohem Grad deutliche Performancegewinne zeigt.

Konkret ist ein Polynom also definiert durch:

```haskell
-- |Polynome sind Listen von Monomen, welche durch Paare (In,a)
-- dargestellt werden. An erster Stelle steht der Grad, an zweiter der
-- Koeffizient.
data Polynom a = PMS { unPMS :: [(Int,a)], clean :: Bool } deriving ()
```

GalFld/Core/Polynomials.hs

Um diese Darstellung *dense* nennen und mit ihr effizient arbeiten zu können, legen wir folgende Beschränkungen für die Implementierung fest:

Invariante 2.1 Für `PMS` L `True` gilt stets, dass die Monome in L alle nicht Null sind und ihrem Grade nach in absteigender Reihenfolge sortiert sind, d.h.

1. für alle $(i, x) \in L$ ist $x \neq 0$.
2. für alle $(i, x), (j, y) \in L$ gilt: Steht (i, x) vor (j, y), so ist $i > j$.

Ein Polynom, das diese Eigenschaften erfüllt, wollen wir auch *wohlgeformt* oder *korrekt dargestellt* nennen.

Beispiel 2.2 Für das Polynom $f(X) = X^5 + 3X^2 + 1$ wäre

```haskell
PMS [(5,1), (2,3), (0,1)] True
```

[1] Was genau eine „Liste“ in der jeweilig benutzten Sprache bedeuten soll, bleibt der Interpretation überlassen.

die korrekte Darstellung.

Damit diese Invariante stets sichergestellt ist, existiert die Funktion `cleanP`, mit der eine `[(Int,a)]` Liste in die korrekte Form gebracht werden kann:

GalFld/Core/Polynomials.hs

```
-- |Lösche (i,0) Paare und sortiere dem Grade nach absteigend
cleanP :: (Num a, Eq a) ⇒ Polynom a → Polynom a
cleanP f@(PMS ms True)  = f
cleanP   (PMS ms False) = PMS (clean' ms) True
  where clean' ms = filter (λ(_,m) → m≠0) $ sortBy (flip (comparing fst)) ms
```

2.1.2 Instanzen

Es wurden die offensichtlichen Instanzen implementiert, d.h. `Eq` und `Num`. Zudem wurden zur Anzeige von Polynomen `Show` und `ShowTex`, zur binären Speicherung `Binary` und für eine Auswertung trotz Lazyness `NFData` implementiert.

Alle auftauchenden Funktionen werden im weiteren Verlauf näher erläutert.

Eq

GalFld/Core/Polynomials.hs

```
instance (Eq a, Num a) ⇒ Eq (Polynom a) where
  f ≡ g = eqP f g
```

Num

GalFld/Core/Polynomials.hs

```
instance (Num a, Eq a) ⇒ Num (Polynom a) where
  {-# INLINE (+) #-}
  f@(PMS _ _) + g@(PMS _ _) = PMS hs True
    where hs = addPM (unPMS $ cleanP f) (unPMS $ cleanP g)
  {-# INLINE (-) #-}
  f@(PMS _ _) - g@(PMS _ _) = PMS hs True
    where hs = subtrPM (unPMS $ cleanP f) (unPMS $ cleanP g)
  {-# INLINE (*) #-}
  f@(PMS _ _) * g@(PMS _ _) = PMS hs True
    where hs = multPM (unPMS $ cleanP f) (unPMS $ cleanP g)
  fromInteger i      = PMS [(0,fromInteger i)] True
  abs _              = error "Prelude.Num.abs: inappropriate abstraction"
  signum _           = error "Prelude.Num.signum: inappropriate abstraction"
  negate (PMS ms b) = PMS ((map . A.second) negate ms) b
```

Show

GalFld/Core/Polynomials.hs

```
instance (Show a, Eq a, Num a) ⇒ Show (Polynom a) where
  show (PMS [] _) = "0"
  show (PMS ms True) = show' $ tuple2List ms
    where show' ms = intercalate "+" $
                      (λss → [s | s ← reverse ss , s ≠ ""]) $
                      zipWith (curry show'') ms [0..]
          show'' :: (Show a, Eq a, Num a) ⇒ (a,Int) → String
          show'' (0,_) = ""
          show'' (m,0) = show m
          {-show' (1,i) = showExp i-}
          show'' (m,i) = show m ⧺ "·" ⧺ showExp i
          showExp :: Int → String
          showExp 0 = ""
          showExp 1 = "\x1B[04mX\x1B[24m"
          showExp i = "\x1B[04mX" ⧺ showExp' (show i) ⧺ "\x1B[24m"
          showExp' :: String → String
          showExp' ""     = []
          showExp' (c:cs) = newC : showExp' cs
            where newC | c ≡ '0' = '⁰'
                       | c ≡ '1' = '¹'
                       | c ≡ '2' = '²'
                       | c ≡ '3' = '³'
                       | c ≡ '4' = '⁴'
                       | c ≡ '5' = '⁵'
                       | c ≡ '6' = '⁶'
                       | c ≡ '7' = '⁷'
                       | c ≡ '8' = '⁸'
                       | c ≡ '9' = '⁹'
  show f   = show $ cleanP f
```

ShowTex

GalFld/Core/Polynomials.hs

```
instance (ShowTex a, Num a, Eq a) ⇒ ShowTex (Polynom a) where
  showTex (PMS [] _) = "0"
  showTex (PMS ms True) = show' $ tuple2List ms
    where show' ms = intercalate "+" $
                      (λss → [s | s ← reverse ss , s ≠ ""]) $
                      zipWith (curry showTex') ms [0..]
          showTex' :: (ShowTex a, Eq a, Num a) ⇒ (a,Int) → String
          showTex' (0,_) = ""
          showTex' (m,i) = showTex m ⧺ showExp i
          showExp :: Int → String
          showExp 0 = ""
          showExp 1 = "\\cdot{}X"
          showExp i = "\\cdot{}X^{" ⧺ show i ⧺ "}"
  showTex f = showTex $ cleanP f
```

2.1.3 Polynome erstellen

Allgemein Invariante 2.1 erfordert auch, dass der Konstruktor des Polynomdatentyps nicht öffentlich gemacht wird und wir benötigen separate Funktionen, um Polynome zu erstellen. Diese sind selbsterklärend:

```haskell
-- |Erzeugt ein Polynom aus einer Liste von Koeffizienten
pList :: (Num a, Eq a) => [a] -> Polynom a
pList ms = PMS (list2TupleSave ms) True
```

GalFld/Core/Polynomials.hs

```haskell
-- |Erzeugt ein Polynom aus einer Liste von Monomen
pTup :: (Num a, Eq a) => [(Int,a)] -> Polynom a
pTup ms = cleanP $ PMS ms False
```

GalFld/Core/Polynomials.hs

Ferner existiert noch eine Variante der „unsicheren“ Erstellung von Polynomen, die eine korrekte Darstellung nach Invariante 2.1 voraussetzt, diese jedoch nicht prüft.

```haskell
-- |Erzeugt ein Polynom aus einer Liste von Monomen
--  Unsichere Variante: Es wird angenommen, dass die Monome
--  in dem Grade nach absteigend sortierter Reihenfolge auftreten!
pTupUnsave :: [(Int,a)] -> Polynom a
pTupUnsave ms = PMS ms True
```

GalFld/Core/Polynomials.hs

Polynome dekonstruieren Den Weg rückwärts zu gehen ist natürlich auch möglich, was `p2Tup` und `p2List` bewerkstelligen:

```haskell
p2Tup :: (Num a, Eq a) => Polynom a -> [(Int,a)]
p2Tup = unPMS . cleanP
```

GalFld/Core/Polynomials.hs

```haskell
p2List :: (Num a, Eq a) => Polynom a -> [a]
p2List = tuple2List . unPMS . cleanP
```

GalFld/Core/Polynomials.hs

Spezielle Polynome Eines der am häufigsten verwendeten Polynome ist das Nullpolynom. Daher gibt es sowohl eine Prüfung, ob ein Polynom null ist, als auch das Nullpolynom selbst als Objekt:

```haskell
-- |Das Nullpolynom
nullP = PMS [] True
```

GalFld/Core/Polynomials.hs

```haskell
{-# INLINE isNullP #-}
isNullP (PMS ms _) = isNullP' ms
isNullP' []         = True
isNullP' ((i,m):ms) | m /= 0     = False
                    | otherwise = isNullP' ms
```

GalFld/Core/Polynomials.hs

Des Weiteren haben wir eine kleine Schreibhilfe zur Erstellung von konstanten Polynomen generiert:

```haskell
-- |Erzeugt ein konstantes Polynom, d.h. ein Polynom von Grad 0
pKonst :: (Eq a, Num a) ⇒ a → Polynom a
pKonst x | x ≡ 0      = nullP
         | otherwise = PMS [(0,x)] True
```

GalFld/Core/Polynomials.hs

2.1.4 Einwertige Operationen auf Polynomen

Der Grad Der Grad eines Polynoms, lässt sich aufgrund Invariante 2.1 sehr leicht herausfinden. Es gilt jedoch zu beachten, dass der Grad des Nullpolynoms nicht 0 ist. Wir haben uns daher entschieden, den Grad als `Maybe Int` zu implementieren:

```haskell
{-# INLINE degP #-}
-- |Gibt zu einem Polynom den Grad
degP :: (Num a, Eq a) ⇒ Polynom a → Maybe Int
degP f@(PMS [] _)    = Nothing
degP (PMS ms True)   = Just $ fst $ head ms
degP f               = degP $ cleanP f
```

GalFld/Core/Polynomials.hs

Es ist klar, dass man meistens einen `Int` als Grad haben möchte, daher haben wir folgende Funktion implementiert:

```haskell
{-# INLINE uDegP #-}
uDegP :: (Num a, Eq a) ⇒ Polynom a → Int
uDegP = fromJust . degP
```

GalFld/Core/Polynomials.hs

Auswerten Natürlich muss man auch etwas in ein Polynom einsetzen können, was wir mit Hilfe des Horner-Schemas (vgl. [18]) implementiert haben. Dies zeigt eine schöne Anwendung der Haskellfunktion `foldl`:

```haskell
{-# INLINE evalP #-}
-- |Nimmt einen Wert und ein Polynom und wertet es dort aus.
--  Mittels Horner Schema
evalP :: (Eq a, Num a) ⇒ a → Polynom a → a
evalP x f = evalP' x (unPMS $ cleanP f)
evalP' :: (Num a) ⇒ a → [(Int,a)] → a
evalP' x []    = 0
evalP' x fs    = snd $ foldl' (λ(i,z) (j,y) → (j,z*x^(i-j)+y)) (head fs) (tail fs)
```

GalFld/Core/Polynomials.hs

Normieren Über das Normieren braucht man nicht viele Worte verlieren.

```haskell
moniP :: (Num a, Eq a, Fractional a) ⇒ Polynom a → Polynom a
moniP f@(PMS [] _)     = f
moniP f@(PMS ms True) = PMS ns True
  where ns = map (λ(i,m) → (i,m/l)) ms
        l  = snd $ head ms
moniP f                = moniP $ cleanP f
```

GalFld/Core/Polynomials.hs

Da man in vielen Situationen das Inverse des Leitkoeffizienten des Polynoms bei der Normierung erhalten möchte, gibt es noch die folgende Variante der Normierung:

```haskell
-- |Normiert f und gibt gleichzeitig das Inverse des Leitkoeffizienten zurück
moniLcP :: (Num a, Eq a, Fractional a) => Polynom a -> (a,Polynom a)
moniLcP f@(PMS [] _)    = (0,f)
moniLcP f@(PMS ms True) = (1,PMS ns True)
  where ns = map (\(i,m) -> (i,m*l)) ms
        l  = recip $ snd $ head ms
moniLcP f               = moniLcP $ cleanP f
```

GalFld/Core/Polynomials.hs

Formale Ableitung

```haskell
-- |Nimmt ein Polynom und leitet dieses ab.
deriveP :: (Num a, Eq a) => Polynom a -> Polynom a
deriveP (PMS [] _) = PMS [] True
deriveP (PMS ms b) = PMS (deriveP' ms) b
  where deriveP' [] = []
        deriveP' ((i,m):ms) | j<0       = deriveP' ms
                            | c==0      = deriveP' ms
                            | otherwise = (j,c) : deriveP' ms
          where j=i-1
                c=m*fromInteger (fromIntegral i)
```

GalFld/Core/Polynomials.hs

Das reziproke Polynom

Definition 2.3 (reziprokes Polynom) Sei $f(X) = \sum_{i=0}^{n} a_i X^i \in R[X]$ für einen Körper R, so ist das *reziproke Polynom von Ordnung d von $f(X)$* für $d \geq n$ gegeben durch

$$f_d^*(X) := X^d f(\tfrac{1}{X}).$$

```haskell
reciprocP2 :: (Eq a, Fractional a) => Int -> Polynom a -> Polynom a
reciprocP2 k f = cleanP $ PMS ms False
  where d  = uDegP f
        ms = map (A.first (k -)) $ unPMS f
```

GalFld/Core/Polynomials.hs

Das reziproke Polynom, wie man es normalerweise kennt, ist dann für $d = n$ in obiger Definition gegeben durch

```haskell
{-# INLINE reciprocP #-}
reciprocP :: (Eq a, Fractional a) => Polynom a -> Polynom a
reciprocP f = reciprocP2 d f
  where d  = uDegP f
```

GalFld/Core/Polynomials.hs

Multiplikation mit Monomen Es ist klar, dass die Multiplikation eines Polynoms mit einem Monom einfacher ist, als der allgemeine Fall. Daher verdient dieses Vorgehen eine eigene Funktion:

GalFld/Core/Polynomials.hs

```haskell
{-# INLINE multMonomP #-}
-- |Multipliziert f mit x^i
multMonomP ∷ (Eq a, Num a) ⇒ Int → Polynom a → Polynom a
multMonomP i (PMS ms b)   = PMS (map (A.first (+i)) ms) b
```

2.1.5 Zweiwertige Operationen auf Polynomen

Gleichheit Bekanntlich sind zwei Polynome genau dann gleich, wenn ihre Koeffizienten übereinstimmen:

GalFld/Core/Polynomials.hs

```haskell
{-# INLINE eqP #-}
eqP ∷ (Eq a, Num a) ⇒ Polynom a → Polynom a → Bool
eqP (PMS ms True) (PMS ns True)  = eqP' ms ns
  where eqP' [] ns = isNullP' ns
        eqP' ms [] = isNullP' ms
        eqP' ((i,m):ms) ((j,n):ns) =  i≡j && m≡n && eqP' ms ns
eqP f g  = eqP (cleanP f) (cleanP g)
```

Addition Hier kommt zum ersten Mal ein kleiner Nachteil der *dense* Darstellung zu Tage, da das Addieren zweier Polynome nicht einfach das elementweise Summieren zweier Listen ist, sondern stets geprüft werden muss, bei welchem Grad man gerade ist:

GalFld/Core/Polynomials.hs

```haskell
{-# INLINE addPM #-}
-- |Addiert Polynome in Monomdarstellung, d.h
--  [(Int,a)] wobei die Liste in Int ABSTEIGEND sortiert ist
addPM ∷ (Eq a,Num a) ⇒ [(Int,a)] → [(Int,a)] → [(Int,a)]
addPM [] gs          = gs
addPM fs []          = fs
addPM ff@((i,f):fs) gg@((j,g):gs)
  | i≡j && c≠0  = (i,c) : addPM fs gs
  | i≡j && c≡0  = addPM fs gs
  | i<j         = (j,g) : addPM ff gs
  | i>j         = (i,f) : addPM fs gg
   where !c = f+g
```

`addPM` darf offensichtlich nur ausgeführt werden, wenn die beiden Polynome Invariante 2.1 erfüllen. Darüber hinaus stellt obige Funktion auch sicher, dass besagte Invariante erhalten bleibt.

Subtraktion Da die funktionale Programmierung lediglich nicht veränderbare Objekte (*immutable objects*) vorsieht, würde durch die Standarddefinition der Subtraktion, nämlich Addition des ersten mit dem negierten zweiten Argument, das zweite Polynom doppelt durchlaufen (einmal beim Negieren und einmal beim Addieren). Um dies zu verhindern, haben wir die Subtraktion separat geschrieben.

GalFld/Core/Polynomials.hs

```
{-# INLINE subtrPM #-}
-- |Subtrahiert Polynome in Monomdarstellung, d.h
--   [(Int,a)] wobei die Liste in Int ABSTEIGEND sortiert ist
subtrPM ∷ (Eq a,Num a) ⇒ [(Int,a)] → [(Int,a)] → [(Int,a)]
subtrPM [] gs          = map (A.second negate) gs
subtrPM fs []          = fs
subtrPM ff@((i,f):fs) gg@((j,g):gs)
  | i≡j && c≠0  = (i,c) : subtrPM fs gs
  | i≡j && c≡0  = subtrPM fs gs
  | i<j         = (j,negate g) : subtrPM ff gs
  | i>j         = (i,f) : subtrPM fs gg
   where !c = f-g
```

Wiederum darf obige Subtraktion nur auf wohlgeformte Polynome angewandt werden.

Multiplikation Es hat sich herausgestellt, dass in den meisten Fällen die „Standardmultiplikation“ die effizienteste ist. Diese ist wie folgt implementiert:

GalFld/Core/Polynomials.hs

```
{-# INLINE multPM #-}
-- | Multiplikation von absteigend sortierten [(Int,a)] Listen
multPM ∷ (Eq a, Num a) ⇒ [(Int,a)] → [(Int,a)] → [(Int,a)]
multPM f [] = []
multPM [] f = []
multPM ms ns = foldr1 addPM summanden
  where  summanden = [multPM' i m ns | (i,m) ← ms]
```

GalFld/Core/Polynomials.hs

```
{-# INLINE multPM' #-}
multPM' i m []                      = []
multPM' i m ((j,n):ns) | c ≡ 0      = multPM' i m ns
                       | otherwise = (k,c) : multPM' i m ns
  where !c = n*m
        !k = i+j
```

Wir haben jedoch auch die Multiplikation nach Karatsuba und eine Multiplikation auf FFT-Grundlage implementiert, wie in Unterabschnitt 2.2.1 nachzulesen ist.

Division mit Rest Wie auch schon bei der Multiplikation von Polynomen, kennt man bei der Division mit Rest verschiedene Algorithmen. Als erste und einfachste Wahl bietet sich die Division mit Rest nach Grundschulmethode an. Diese hat sich jedoch als langsamste erwiesen und wurde daher wieder aus dem Code entfernt. Die nun in den meisten Fällen schnellste Methode ist die Division mit Hilfe des Horner-Schemas. Eine sehr gute und ausführliche Erklärung findet sich in [20].

Wiederum lässt sich die Division per Horner-Schema sehr schön rekursiv in Haskell niederschreiben.

GalFld/Core/Polynomials.hs

```
{-# INLINE divPHornerM' #-}
-- |Horner für absteigend sortierte [(Int,a)] Paare
divPHornerM' _  [] _ _ = []
divPHornerM' divs ff@((i,f):fs) lc n
  | n > fst (head ff)  = ff
  | otherwise              = (i,fbar) : divPHornerM' divs hs lc n
  where fbar = f/lc
        {-# INLINE hs #-}
        hs   = addPM fs $! js
        {-# INLINE js #-}
        js   = map ( (+) (i-n) A.*** (*) fbar) divs
```

Wie in den obigen Funktionen kommt man auch hier nicht ohne Overhead aus, der notwendig ist, um die verschiedenen Polynom-Status (`cleanP` betreffend) zu behandeln und die initialen Parameter festzulegen.

GalFld/Core/Polynomials.hs

```
-- |divP mit Horner Schema
--  siehe http://en.wikipedia.org/wiki/Synthetic_division
divP ∷ (Show a, Eq a, Fractional a) ⇒
                              Polynom a → Polynom a → (Polynom a,Polynom a)
divP = divPHorner
```

GalFld/Core/Polynomials.hs

```
divPHorner a (PMS [] _)    = error "Division by zero"
divPHorner a@(PMS as True) b@(PMS bs True)
    | isNullP a        = (PMS [] True,PMS [] True)
    | degDiff ≤ 0      = (PMS [] True,a)
    | otherwise        = toP $ A.first (map (A.first (λi → i-degB))) $
                                                     splitAt splitPoint horn
  where horn       = divPHornerM' bs as lc degB
        degDiff    = uDegP a - uDegP b + 1
        bs         = tail $ unPMS $ negate b
        as         = unPMS a
        lc         = getLcP b
        degB       = uDegP b
        splitPoint = length [i | (i,j) ← horn, i ≥ degB]
        toP (a,b)  = (PMS a True, PMS b True)
divPHorner a b = divPHorner (cleanP a) (cleanP b)
```

„Division“ Für den Fall, dass man bereits weiß, dass ein Polynom durch ein anderes teilbar ist, haben wir den Operator `(@/)` definiert [2]. Dieser ist selbstredend nichts an-

[2] Zu beachten ist hierbei, dass die Benutzung von `(/)` nicht möglich ist, da es eine `Fractional`-Instanz erfordern würde, die es auf Polynomen ja offenbar nicht gibt.

deres als Division mit Rest, wobei lediglich der erste Eintrag des Tupels zurückgegeben wird.

Modulo Dual zu `(@/)` ist `modByP`, das einfach den zweiten Eintrag von `divP` liefert:

```
{-# INLINE modByP #-}
-- |Nimmt ein Polynom und rechnet modulo ein anderes Polynom.
--  Also Division mit Rest und Rückgabe des Rests.
modByP :: (Show a, Eq a, Fractional a) ⇒ Polynom a → Polynom a → Polynom a
modByP f p = snd $ divP f p
```

GalFld/Core/Polynomials.hs

Erweiterter Euklidischer Algorithmus Auch der erweiterte Euklidische Algorithmus basiert auf `divP`. Er ist – selbstverständlich rekursiv – hier gegeben durch:

```
{-# INLINE eekP #-}
-- |Erweiterter Euklidischer Algorithmus: gibt (d,s,t) zurück mit
--  ggT(a,b) = d = s*a + t*b
eekP :: (Show a, Eq a, Fractional a) ⇒ Polynom a → Polynom a
                                          → (Polynom a, Polynom a, Polynom a)
eekP f g | g ≡ 0     = (moniP f, PMS [(0,recip $ getLcP f)] True, PMS [] True)
         | otherwise = (d,t,s-t*q)
  where (q,r)   = divP f g
        (d,s,t) = eekP g r
```

GalFld/Core/Polynomials.hs

Größter gemeinsamer Teiler Aus dem erweiterten Euklidischen Algorithmus erhält man selbstverständlich auch den ggT zweier Polynome:

```
{-# INLINE ggTP #-}
-- |Algorithmus für ggT
ggTP :: (Show a, Eq a, Fractional a) ⇒ Polynom a → Polynom a → Polynom a
ggTP f g = (λ (x,_,_) → x) $ eekP f g
```

GalFld/Core/Polynomials.hs

2.1.6 Weiteres

Nullstellensuche Möchte man prüfen, ob ein Polynom in einer gewissen Menge von Elementen eine Nullstelle besitzt, so ist dies mit folgender Funktion möglich.

```
hasNs :: (Eq a, Fractional a) ⇒ Polynom a → [a] → Bool
hasNs f es = not (null [f | e ← es, evalP e f ≡ 0])
```

GalFld/Core/Polynomials.hs

Auflisten aller Polynome Folgende Funktion listet alle monischen Polynome auf, deren Grad in der Liste `[Int]` vorkommt und deren Koeffizienten in der Liste `[a]` liegen.

```
getAllMonicPs :: (Num a, Fractional a, Eq a) ⇒ [a] → [Int] → [Polynom a]
getAllMonicPs es is = map (`PMS` True) $ concat [allMonics i | i ← is]
  where allMonics 0 = [[(0,1)]]
        allMonics i = [(i,1)] : [(i,1):rs | rs ← ess (i-1)]
        ess i       | i ≡ 0     = [[(0,y)] | y ← swpes]
                    | otherwise = [[(i,y)] | y ← swpes] ⧺ ess (i-1) ⧺
                                  [(i,y):ys | y ← swpes, ys ← ess (i-1)]
        swpes       = filter (≠ 0) es
```

GalFld/Core/Polynomials.hs

Möchte man Polynome von Grad 0 bis zu einem gewissen Grad, so liefert dies die Funktion `getAllMonicP`.

```
getAllMonicP :: (Num a, Fractional a, Eq a) ⇒ [a] → Int → [Polynom a]
getAllMonicP es d = getAllMonicPs es [0..d]
```

GalFld/Core/Polynomials.hs

Zuletzt kann man sich natürlich noch zusätzlich die nicht-monischen Polynome ausgeben lassen; wie oben in beiden Varianten (d.h. Grade als Liste oder als Obergrenze gegeben).

```
-- |Nimmt eine Liste und eine Liste von Grade und erzeugt daraus alle
-- Polynome, deren Grade in der Liste enthalten sind.
getAllPs :: (Num a, Fractional a, Eq a) ⇒ [a] → [Int] → [Polynom a]
getAllPs es ds = [PMS (map (A.second (e*))$ unPMS f) True
                   | f ← getAllMonicPs es ds , e ← es , e ≠ 0]
```

GalFld/Core/Polynomials.hs

```
-- |Nimmt eine Liste und Grad und erzeugt daraus alle Polynome bis zu diesem
-- Grad.
-- Das Nullpolynom (P[]) ist NICHT enthalten.
getAllP :: (Num a, Fractional a, Eq a) ⇒ [a] → Int → [Polynom a]
getAllP es d = [PMS (map (A.second (e*)) $ unPMS f) True | f ← getAllMonicP es d
                                        , e ← es , e ≠ 0]
```

GalFld/Core/Polynomials.hs

Conway-Polynome Die *Conway-Polynome* bieten in gewisser Weise eine kanonische Möglichkeit Körpererweiterungen endlicher Körper zu charakterisieren. Für Definitionen und Eigenschaften sei auf `http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/index.html` verwiesen, wo auch die Conway-Polynome zu finden sind, die wir in der Datei `GalFld/Core/Polynomials/Conway.hs` hinterlegt haben.

2.2 Alternative Polynomalgorithmen

2.2.1 Verschiedene Multiplikationsalgorithmen

Karatsuba

Einer der häufigsten Multiplikationsalgorithmen für Polynome ist sicher der Karatsuba. Er basiert auf der Idee, Multiplikationen durch Additionen zu ersetzen, die im Allgemeinen „billiger“ sind.

Der Basisfall für die Multiplikation zweier Polynome von Grad 1 lässt die Idee des Algorithmus deutlich werden:

$$(a_1X + b_1) \cdot (a_2X + b_2) = AX^2 + (C - A - B)X + B$$

wobei

$$A = a_1b_1, \qquad B = a_2b_2, \qquad C = (a_1 + b_1)(a_2 + b_2)\,.$$

Damit können die ursprünglich 4 auftretenden Multiplikationen des Standardalgorithmus durch 3 Multiplikationen und 4 Additionen ersetzt werden. Dies lässt sich natürlich rekursiv anwenden. Da die Polynome jedoch nicht immer gleichen Grades sind und dieser selten eine Zweierpotenz ist (letzteres ist notwendig, damit der Algorithmus rekursiv bis zu obigem Grundfall laufen kann), wählt man die nächstkleinere Zweierpotenz des Maximums der beiden Grade als Teilungspunkt für den rekursiven Aufruf. Die Implementierung erfolgt dabei in drei Schritten:

```haskell
{-# INLINE multPK #-}
multPK :: (Show a, Num a, Eq a) => Polynom a -> Polynom a -> Polynom a
multPK f g = PMS h True
  where h = multPMKaratsuba ((unPMS.cleanP) f) ((unPMS.cleanP) g)
```

GalFld/Core/Polynomials.hs

Hierzu gibt es nichts zu sagen. Im nächsten Schritt wird dann die passende Zweierpotenz ermittelt und damit der eigentliche Karatsuba aufgerufen:

```haskell
{-# INLINE multPMKaratsuba #-}
multPMKaratsuba :: (Show a, Num a, Eq a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]
multPMKaratsuba f g  = multPMK' n f g
  where n  = next2Pot (max df dg) `quot` 2
        df = if null f then 0 else fst (head f) + 1
        dg = if null g then 0 else fst (head g)+ 1
```

GalFld/Core/Polynomials.hs

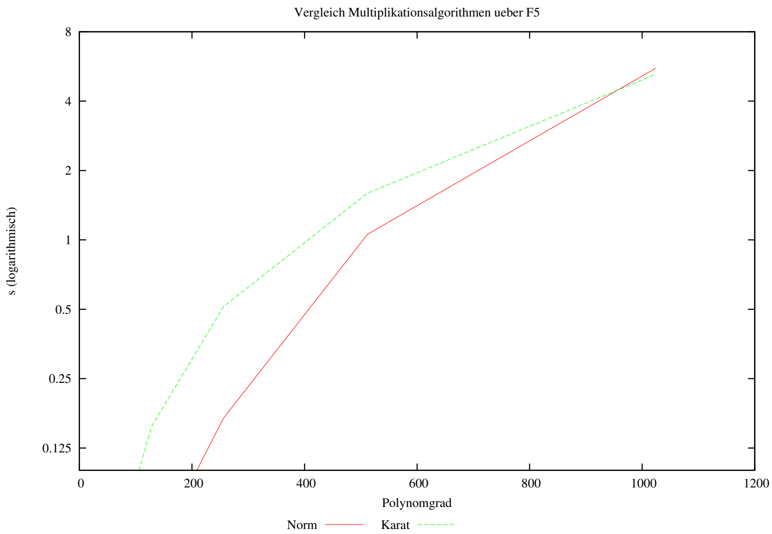
Letztlich bleibt nur der Algorithmus übrig. Aufgrund der Tupeldarstellung der Polynome, ist die Trennung selbiger nicht so einfach und elegant, wie es die Listendarstellung erlaubt hätte. Nichtsdestotrotz ist diese Implementierung auch in diesem Fall effizienter.

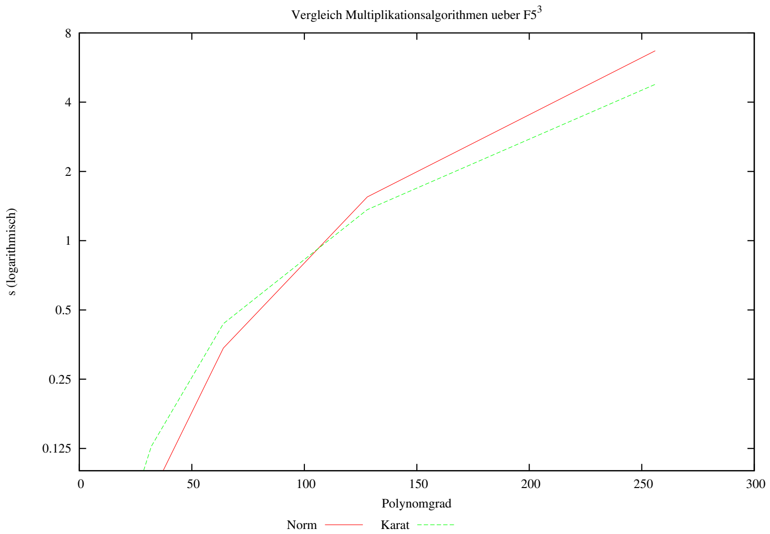
GalFld/Core/Polynomials.hs

```
-- Der eigentliche Karatsuba
multPMK' :: (Show a, Num a, Eq a) ⇒ Int → [(Int,a)] → [(Int,a)] → [(Int,a)]
multPMK' _ _ [] = []
multPMK' _ [] _ = []
multPMK' _ [(i,x)] g = map ((+) i A.*** (*) x) g
multPMK' _ f [(i,x)] = map ((+) i A.*** (*) x) f
multPMK' 1 [(i1,x1),(i2,x2)] [(j1,y1),(j2,y2)]
      = [(2,p1), (1,p3-p1-p2), (0,p2)]
  where !p1 = x1*y1
        !p2 = x2*y2
        !p3 = (x1+x2)*(y1+y2)
multPMK' n f g = addPM e1 $ addPM e2 e3
  where  -- High und Low Parts
        {-# INLINE fH' #-}
        fH' = takeWhile (λ(i,_) → i≥n) f
        {-# INLINE fH #-}
        fH = map (A.first (λi → i-n)) fH'
        {-# INLINE fL #-}
        fL = f \\ fH'
        {-# INLINE gH' #-}
        gH' = takeWhile (λ(i,_) → i≥n) g
        {-# INLINE gH #-}
        gH = map (A.first (λi → i-n)) gH'
        {-# INLINE gL #-}
        gL = g \\ gH'
        -- Rekursiver Karatsuba
        {-# INLINE p1 #-}
        p1 = multPMK' (n `quot` 2) fH gH
        {-# INLINE p2 #-}
        p2 = multPMK' (n `quot` 2) fL gL
        {-# INLINE p3 #-}
        p3 = multPMK' (n `quot` 2) (addPM fH fL) (addPM gH gL)
        {-# INLINE e1 #-}
        e1 = map (A.first (+(2*n))) p1
        {-# INLINE e2 #-}
        e2 = map (A.first (+n)) $ subtrPM p3 (addPM p1 p2)
        {-# INLINE e3 #-}
        e3 = p2
```

Ein kleiner Vergleich Auf Polynomen über den `PrimeFields` bringt der Karatsuba erst bei sehr hohen Graden einen leichten Vorteil gegenüber der Standardmultiplikation. Betrachten wir jedoch ein Beispiel über einem Erweiterungskörper, so kann der Karatsuba seinen Vorteil ausspielen, da dort ja die Koeffizienten selbst Polynome sind, und daher Addition weitaus „billiger“ ist als Multiplikation. Abbildung 2.1 zeigt dies deutlich.

Abbildung 2.1: Vergleich von normaler Multiplikation mit Karatsuba





FFT-Multiplikation: Der Schönhage-Strassen-Algorithmus

Eine der Idee nach weitaus komplexere Möglichkeit Polynome zu multiplizieren, ist die Multiplikation auf Basis der schnellen Fourier-Transformation (engl. *fast Fourier transform* (FFT)). Sie ist die bislang schnellste bekannte Methode. Allerdings gilt dies nur für die asymptotische Laufzeit! Daher konnten wir leider nur feststellen, dass die FFT-Multiplikation stets weitaus langsamer ist, als der Standardalgorithmus oder Karatsuba.

Die Idee der FFT-Multiplikation basiert auf der Tatsache, dass sich das Produkt zweier Polynome in diskreter Fourier-Transformation (DFT) als komponentenweises Produkt der Fourier-Transformierten beider Polynome berechnen lässt. Dies wollen wir uns näher betrachten:

Im Folgenden sei stets R ein kommutativer Ring mit Eins und $\omega \in R$ eine primitive n-te Einheitswurzel.

Definition 2.4 Ist $f \in R[X]$ ein Polynom, so ist seine *diskrete Fourier-Transformation (DFT)* definiert als

$$f^\wedge(\omega) = (f(\omega^j) : \ j = 0, \dots, n-1) \in R^n \,.$$

Eine wesentliche Eigenschaft liefert folgende Proposition.

Proposition 2.5 *Sei $f \in R[X]$ und $F := f^\wedge(\omega)$ seine DFT. Ist $n \in R$ eine Einheit, so gilt*

$$f(X) = (\tfrac{1}{n} F^\wedge(\omega^{-1}))(X) \,.$$

Beweis. [16, Proposition 4.9]. □

Die auftauchende Frage zur Notation eines n-Tupels als Polynom beantwortet nachstehende Definition:

Definition 2.6 Sei $r = (r_0, \dots, r_{n-1}) \in R^n$, so definieren wir

$$r(X) := \sum_{i=0}^{n-1} r_i X^i \ \in R[X] \,.$$

Ferner notieren wir für $f(X) = \sum_{i=0}^{n-1} f_i X^i \in R[X]$

$$f = (f_0, \dots, f_{n-1}) \in R^n \,.$$

Die DFT eines Polynoms lässt sich sehr schnell durchführen, wenn man $n = 2^l$ eine Zweierpotenz setzt:

Algorithmus 2.1: FFT

Input: $n = 2^l$, $\omega \in R$ eine primitive $n-$te Einheitswurzel, $f \in R^n$
Output: $f^\wedge(\omega)$
Algorithmus FFT(n,ω,f):
1. Setze
   $r := \left(f_j + f_{j+\frac{n}{2}} : \ j = 0, \ldots, \frac{n}{2}\right)$
   $r_\omega := \left(\omega^j (f_j - f_{j+\frac{n}{2}}) : \ j = 0, \ldots, \frac{n}{2}\right)$
2. Berechne rekursiv $a :=$ FFT($\frac{n}{2}$, ω^2, r) und $b :=$ FFT($\frac{n}{2}$, ω^2, r_ω)
3. Mische die Ergebnisse: $f^\wedge(\omega) := (a_0, b_0, a_1, \ldots, a_{\frac{n}{2}-1}, b_{\frac{n}{2}-1})$

Wie man damit Polynome multiplizieren kann, erklärt nachstehender Algorithmus:

Algorithmus 2.2: FFT-Multiplikation

Input: $\omega \in R$ eine primitive $n-$te Einheitswurzel, $n = 2^l$,
$f(X), g(X) \in R[X]$ mit $\deg f + \deg g < n$
Output: $h(X) = f(X)g(X) \in R[X]$
Algorithmus FFTM(f,g,n,ω):
1. Berechne $a := f^\wedge(\omega)$, $b := g^\wedge(\omega)$.
2. Berechne komponentenweise $c := a \odot b$.
3. Setze $h(X) := (\frac{1}{n} c^\wedge(\omega^{-1}))(X)$

Es bleiben ein paar Probleme offen: Um mit obigem Algorithmus Polynome zu multiplizieren, muss in R für jede Zweierpotenz n eine primitive n-te Einheitswurzel existieren und 2 eine Einheit sein. Die Tatsache, dass 2 eine Einheit sein muss, spielt für den Fall der Anwendung – nämlich Multiplikation von Polynomen über endlichen Körpern – quasi keine Rolle, da für Charakteristik ungleich 2 dies dort ja immer gegeben ist.[3] Jedoch bleibt offen, wie man eine n-te Einheitswurzel finden soll. Die allgemeine Antwort lautet in diesem Fall: Wir suchen gar nicht, sondern modifizieren das Setting so, dass stets eine n-te Einheitswurzel gegeben ist:

Lemma 2.7 *Sei R ein kommutativer Ring und $2 \in R$ eine Einheit. Sei ferner $n = 2^l$ mit $l \geq 1$. Dann ist*

$$\omega := X \ \in \ R_n := R[X]/(X^n + 1)$$

eine primitive $(2n)$-te Einheitswurzel.

Beweis. [16, Lemma 4.15]. □

[3] Für Charakteristik 2 existieren ohnehin spezielle Algorithmen auf Basis von Binäroperationen!

Das bedeutet, wir „erzwingen" die Existenz einer passenden primitiven Einheitswurzel. Damit bleibt nur noch die Frage, wie wir ein Polynom $f(X) \in R[X]$ als bivariates Polynom in $R[X,Y]/(Y^n+1)$ lesen können, um dort die FFT-Multiplikation anwenden zu können. Eine Antwort und das konkrete Vorgehen gibt der Schönhage-Strassen-Algorithmus:

Algorithmus 2.3: Schönhage-Strassen-Multiplikation

Input: $f(X), g(X) \in R[X]$, so dass $2 \in R$ eine Einheit
Output: $h(X) = f(X)g(X) \in R[X]$
Algorithmus SS(f,g):

1. Wähle $n = 2^l$ mit $\deg f + \deg g < n$.
2. Setze $m := 2^{\lfloor \frac{l}{2} \rfloor}$ und $m' := \frac{n}{m}$.
3. Zerlege f und g in ,,Blöcke'':
   $f(X) = \sum_{j=0}^{m'-1} f_j(X)X^{mj}$
   $g(X) = \sum_{j=0}^{m'-1} g_j(X)X^{mj}$
4. Setze $R_{2m} := R[X]/(X^{2m}+1)$ und
   $F(X,Y) := \sum_{j=0}^{m'-1} f_j(X)Y^j \ \in R_{2m}[Y]$
   $G(X,Y) := \sum_{j=0}^{m'-1} g_j(X)Y^j \ \in R_{2m}[Y]$
5. Setze $\xi := \begin{cases} X, & l \text{ gerade} \\ X^2, & \text{sonst} \end{cases}$ und $\omega := \xi^2$.
6. Setze $F^*(Y) := F(\xi Y)$, $G^*(Y) := G(\xi Y)$.
7. Berechne $H^*(Y) = \text{FFTM}(F^*, G^*, 2m, \omega)$
8. Setze $H(Y) := H^*(\xi^{-1}Y)$.
9. Setze $h(X) := H(X, X^m) \bmod X^n - 1$.

Satz 2.8 *Algorithmus 2.3 multipliziert zwei Polynome von Grad* $< \frac{n}{2}$ *in*

$$\mathcal{O}(n \log_2 n \log_2 \log_2 n)$$

Ringoperationen.

Beweis. [16, Algorithmus 4.18]. □

Implementierung Zunächst führen wir ein paar kleine Hilfsfunktionen an, die wir in der Implementierung des Schönhage-Strassen-Algorithmus brauchen werden:

GalFld/Core/Polynomials/FFTuple.hs

```haskell
{-# INLINE intersperseL #-}
-- |Intersperse mit 2 Listen
intersperseL :: [a] -> [a] -> [a]
intersperseL ys    []       = ys
intersperseL []    xs       = xs
intersperseL (y:ys) (x:xs)   = y : x : intersperseL ys xs
```

GalFld/Core/Polynomials/FFTTuple.hs

```haskell
{-# INLINE zipWith' #-}
-- wie zipWith. Setzt das Ende der längeren Liste an die kürzere an.
zipWith' :: (t->t->t) -> t -> [t] -> [t] -> [t]
zipWith' _ _ xs [] = xs
zipWith' f t [] ys = map (f t) ys
zipWith' f t (x:xs) (y:ys) = (f x y) : zipWith' f t xs ys
```

GalFld/Core/Polynomials/FFTTuple.hs

```haskell
{-# INLINE log2 #-}
-- |ineffiziente Log 2 Berechnung
log2 :: Int -> Int
log2 0 = 0
log2 1 = 0
log2 n = log2' 1 n
  where log2' i 1 = max 0 (i-1)
        log2' i 2 = i
        log2' i n = 1 + (log2' i $! n `quot` 2)
```

Nun können wir zu den eigentlichen Funktionen übergehen. Wie in der Erklärung beginnen wir mit der Berechnung der FFT nach Algorithmus 2.1.

GalFld/Core/Polynomials/FFTTuple.hs

```haskell
-- |Berechnet die FFT eines Polynoms f.
--  Benötigt eine primitive n-te Einheitswurzel,
--  wobei n eine 2er Potenz ist (Dies wird NICHT überprüft!)
--  Diese wird dargestellt als Funktion w: Int -> a -> a,
--  wobei f(i,x) = w^i*x für die n-te EWL w auswertet.
--
--  vgl. Computer Algebra Algorithmus 4.11
fftP :: (Show a, Num a, Eq a) => (Int -> a -> a) -> Int -> Polynom a -> [a]
fftP w n f = fft w (+) (-) 0 1 n (p2List f)
```

GalFld/Core/Polynomials/FFTTuple.hs

```haskell
-- | Int -> a -> a  Implementierung der primitiven Einheitswurzel
--    a -> a -> a     Addition auf a
--    a -> a -> a     Subtraktion auf a
--    -> a              Die Null
--    -> Int           Aktuelle 2er Potenz (Starte mit 1)
--    -> Int           FFT bis n
--    -> [a]           Eingangsliste
--    -> [a]           Ausgabeliste
fft :: (Show a) => (Int -> a -> a) -> (a->a->a) -> (a->a->a) -> a
                                                   -> Int -> Int -> [a] -> [a]
fft _ _ _ _ _ 1 fs = fs
fft w addF subF zero i n fs = intersperseL ls' rs'
  where !i'  = 2*i
        ls' = fft w addF subF zero i' m ls
        ls  = take m $ zipWith' (addF) zero fs fss
        rs' = fft w addF subF zero i' m rs
        rs  = take m $ zipWith (w) [i | i<-[0..]] $ zipWith' (subF) zero fs fss
        fss = drop m fs
        !m   = n `quot` 2
```

Damit können wir nun Algorithmus 2.3 konkret machen; wiederum zunächst auf Polynomebene und dann auf den eigentlichen Listen:

GalFld/Core/Polynomials/FFTTuple.hs

```
{-# INLINE ssP #-}
-- | Schönhage-Strassen für Polynome
ssP :: (Show a, Fractional a, Num a, Eq a) ⇒ Polynom a → Polynom a → Polynom a
ssP f g | isNullP f || isNullP g = nullP
        | otherwise              = pTup $ ss l fs gs
  where fs = p2Tup f
        gs = p2Tup g
        -- || deg f*g < 2^l ||
        l  = 1 + log2 (uDegP f + uDegP g)
```

GalFld/Core/Polynomials/FFTTuple.hs

```
-- |Der eigentliche Schönhage-Strassen Algorithmus.
--  Funktioniert nur, falls 2 eine Einheit ist!
ss :: (Show a, Num a, Fractional a, Eq a) ⇒ Int → [(Int,a)] → [(Int,a)] → [(Int,a)]
-- ss funktioniert nur für l>2
ss 1 f g = multPM f g
ss 2 f g = multPM f g
ss l f g
  | isNullP' f || isNullP' g = []
  | otherwise = foldr1 (addPM) $
                              reduceModxn (2^l) $ zipWith (multx (m)) [0..] hs
  where -- << n = 2^l = m * m' >>
        !l' = l `quot` 2
        !m  = 2^l'
        !m' = 2^(l-l')
        !fs = ssBuildBlocks (m*(m'-1)) m f
        !gs = ssBuildBlocks (m*(m'-1)) m g
        -- auf FFT vorbereiten
        !fs' = zipWith (multx (xi)) [0..] fs
        !gs' = zipWith (multx (xi)) [0..] gs
        -- FFT durchführen
        !xi    = if odd l then 1 else 2
        !fftFs = reduceModxn (2*m) $ fft (multx (xi*2))
                                              (addPM) (subtrPM) [] 1 m' fs'
        !fftGs = reduceModxn (2*m) $ fft (multx (xi*2))
                                              (addPM) (subtrPM) [] 1 m' gs'
        -- Multiplikation der Ergebnisse und rekursiver Aufruf von ss
        !fftHs = reduceModxn (2*m) $ zipWith (ss (l'+1)) fftFs fftGs
        -- Inverse-FFT
        !hs''  = reduceModxn (2*m) $ fft (multx (xi*(2*m'-2)))
                                            (addPM) (subtrPM) [] 1 m' fftHs
        -- * 1/m'
        !hs'   = map (map (A.second (λx → x / (fromIntegral m')))) hs''
        -- Rückwandlung zu H(x,y)
        !hs    = reduceModxn (2*m) $ zipWith (multx (xi*(2*m'-1))) [0..] hs'
```

`ssBuildBlocks` ist dabei Schritt 3 in Algorithmus 2.3 und gegeben durch

```haskell
{-# INLINE ssBuildBlocks #-}
ssBuildBlocks :: (Show a, Eq a, Num a) ⇒ Int → Int → [(Int,a)] → [[(Int,a)]]
ssBuildBlocks 0 _ fs = [fs]
ssBuildBlocks n m fs = (ssBuildBlocks (n - m) m ns) ⧺ [ms]
  where ms' = filter (λ(i,x) → i ≥ n) fs
        ns  = fs \\ ms'
        ms  = map (A.first (λi → i-n)) ms'
```

GalFld/Core/Polynomials/FFTTuple.hs

Des Weiteren ist eine schnelle Reduktion $\pmod x)^n + 1$ nötig:

```haskell
{-# INLINE reduceModxn #-}
-- | Reduziert die innere Liste modulo x^n+1
reduceModxn :: (Show a, Num a, Eq a) ⇒ Int → [[(Int,a)]] → [[(Int,a)]]
reduceModxn _ [] = []
reduceModxn n x@(xs:xss)
    | l ≥ n      = reduceModxn n $ (hs:xss)
    | otherwise = xs : reduceModxn n xss
  where l   = if null xs then 0 else fst $ head xs
        fs' = filter (λ(i,x) → i≥n) xs
        fs  = map (λ(i,x) → (i-n,negate x)) fs'
        gs  = xs \\ fs'
        hs  = addPM gs fs
```

GalFld/Core/Polynomials/FFTTuple.hs

Zuletzt haben wir noch die Multiplikation mit dem Monom x^{i*j} als separate Funktion gestaltet, die offenbar schneller ist, als der normale Multiplikationsalgorithmus.

```haskell
{-# INLINE multx #-}
-- | Multipliziert mit x^(i*j)
multx :: (Num a) ⇒ Int → Int → [(Int,a)] → [(Int,a)]
multx _ _ []    = []
multx j i xs    = map (A.first (λi→i+k)) xs
  where !k = j*i
```

GalFld/Core/Polynomials/FFTTuple.hs

2.2.2 Division mit Rest mit Inversen modx^l

Im Folgenden stellen wir eine Möglichkeit vor, die Division mit Rest zweier Polynome in genau der gleichen asymptotischen Laufzeit zu bewerkstelligen wie die Multiplikation. Wir halten uns dabei eng an [13] und [1]. Die Idee für diese schöne und zugleich schnelle Methode liefert folgende Proposition.

Proposition 2.9 *Sei $f(X) \in R[X]$ für einen Ring R. Sei $f(0) = 1$ und $l \in \mathbb{N}$. Dann lässt sich $h \in R[X]$ mit*

$$h(X)\, f(X) \quad \equiv \quad 1 \quad \bmod X^l$$

in $\mathcal{O}(m(l))$ berechnen, wobei $m(l)$ die Anzahl der Multiplikationen in R ist, die nötig sind um zwei Polynome in $R[X]$ von Grad l zu multiplizieren.[4]

[4] Man spricht auch von *Multiplikationszeit*, vgl. [13, Definition 2].

Beweis. Betrachte Algorithmus 2.4 und die genauere Analyse im Beweis von [13, Theorem 2]. □

Die konkrete Antwort liefert der folgende Algorithmus.

Algorithmus 2.4: Invertieren $\bmod X^l$

Input: $f(X) \in R[X]$ mit $f(0) = 1$, $l \in \mathbb{N}$
Output: $h(X) \in R[X]$ mit $h(X)\, f(X) \equiv 1 \bmod X^l$
Algorithmus INV_MOD_MONOM(f,l):
1. Setze $g_0 := 1$, $r := \lceil \log_2(l) \rceil$.
2. **for** $i := 1$ **to** l **do**
   $g_i(X) := (2g_{i-1}(X) - f(X)\, g_{i-1}(X)^2) \mod X^{2^i}$
   endfor
3. Setze $h(X) := g_r(X)$.

Bemerkung 2.10 Man beachte, dass in Algorithmus 2.4 stets $g_i \equiv g_{i-1} \bmod X^{2^{i-1}}$ gilt. Das bedeutet, dass man innerhalb der Schleife zur Berechnung von g_i lediglich Polynome von Grad maximal 2^{i-1} multiplizieren muss. Dies sollte man (um ein effizientes Vorgehen sicherzustellen) bei der Implementierung unbedingt beachten.

Nun kann man damit einen Algorithmus zur Division mit Rest aufstellen. (Man erinnere sich kurz an die Definition des reziproken Polynoms von Ordnung d aus Definition 2.3)

Algorithmus 2.5: Division mit Rest durch Invertieren $\bmod X^l$

Input: $a, b \in R[X]$ mit $\deg b \leq \deg a$
Output $q, r \in R[X]$ mit $a = qb + r$
Algorithmus DIV_INV(a,b):
1. Setze $n := \deg a$, $m := \deg b$, $l := n - m + 1$
2. Setze $\bar{b}(X) := \frac{1}{b_m} b(X)$ für b_m den Leitkoeffizienten von b
2. Setze $f(X) := \bar{b}_l^*(X)$
3. Berechne $g(X) :=$ INV_MOD_MONOM(f,l)
4. Berechne $q'(X) := g(X)\, a_l^*(X) \mod X^l$
5. Setze $q(X) := b_m \cdot {q'}^*_{n-m}(X)$ und $r(X) := a(X) - b(X)q(X)$

Satz 2.11 *Algorithmus 2.5 führt die Division mit Rest für $a, b \in R[X]$ mit $n := \deg a$, $m := \deg b$ in $\mathcal{O}(m(\max\{n - m, m\}))$ durch.*

Beweis. [13, Theorem 3]. □

Implementierung

Betrachten wir nun die konkrete Implementierung und beginnen bei Algorithmus 2.5.

```haskell
divPInv :: (Show a, Eq a, Fractional a) ⇒
             Polynom a → Polynom a → (Polynom a, Polynom a)
divPInv a b
    | isNullP a = (nullP, nullP)
    | a ≡ b     = (pKonst 1,nullP)
    | l ≤ 0     = (nullP,a)
    | otherwise = (q',r)
  where n  = uDegP a
        m  = uDegP b
        l  = n-m+1
        (lc,b') = moniLcP b
        f  = reciprocP2 m b'
        g  = invModMonom f l
        q  = multPInter l 0 g $ reciprocP2 n a
        q' = multKonstP lc $ reciprocP2 (l-1) q
        r  = a - b*q'
```

GalFld/Core/Polynomials.hs

`reciprocP2` ist dabei gerade die Berechnung des reziproken Polynoms passender Ordnung, wie bereits oben beschrieben. `multPInter l 0` berechnet dabei das Produkt der Polynome – in diesem Fall – nur bis zum Grad $< l$; liefert also gerade das $\bmod X^l$ in Schritt 4 von Algorithmus 2.5.

```haskell
{-# INLINE multPInter #-}
-- |Multipliziert f mit g, wobei nur Terme mit x^l für
--   l > lLow und l < lHigh betrachtet werden
multPInter :: (Show a, Eq a, Num a) ⇒ Int → Int → Polynom a → Polynom a → Polynom a
multPInter _ _ (PMS [] _) _ = nullP
multPInter _ _ _ (PMS [] _) = nullP
multPInter lHigh lLow f g
      = PMS (multPMInter lHigh lLow ((unPMS.cleanP) f) ((unPMS.cleanP) g)) True
```

GalFld/Core/Polynomials.hs

```haskell
{-# INLINE multPMInter #-}
-- |Multipliziert f mit g, wobei nur Terme mit x^l für
--   l > lLow und l < lHigh betrachtet werden
multPMInter :: (Show a, Eq a, Num a) ⇒ Int → Int →
                                [(Int,a)] → [(Int,a)] → [(Int,a)]
multPMInter _ _ f [] = []
multPMInter _ _ [] f = []
multPMInter lHigh lLow ms ns = foldr1 addPM summanden
  where summanden = [multPMInter'  i m ns | (i,m) ← ms]
        {-# INLINE multPMInter' #-}
        multPMInter' i m [] = []
        multPMInter' i m ((j,n):ns)
          | k < lLow || k ≥ lHigh || c ≡ 0 = multPMInter' i m ns
          | otherwise                      = (k,c) : multPMInter' i m ns
         where !c = n*m
               !k = i+j
```

GalFld/Core/Polynomials.hs

Letztlich bleibt dann noch die Angabe des eigentlichen Invertierens $\bmod X^l$.

GalFld/Core/Polynomials.hs

```
invModMonom :: (Show a, Num a, Eq a, Fractional a) ⇒ Polynom a → Int → Polynom a
invModMonom h k  | isNullP h  = nullP
                 | otherwise  = PMS (invModMonom' [(0,1)] 1) True
  where hs = unPMS $ cleanP h
        invModMonom' !a !l
          | l ≥ k     =  a
          | otherwise = invModMonom' b lnew
          where -- g_i+1 = (2*g_i - h*g_i^2) mod x^(2^i)
                b = map (A.second negate) a' ++ a
                -- a' = h*g_i^2
                a' = multPMInter lnew l hs $ multPMInter lnew 0 a a
                -- nächster Schritt
                lnew = 2*l
```

`multPInter lnew l` stellt – wie in Bemerkung 2.10 erwähnt – sicher, dass man nur die Multiplikationen durchführt, die auch wirklich notwendig sind.

Dazu ein kleines Beispiel.

Beispiel 2.12 Wir wollen $q(X), r(X) \in \mathbb{F}_5[X]$ finden mit $a(X) = b(X)q(X) + r(X)$ für

$$\begin{aligned} a(X) &:= X^5 + 4X^4 + 3X^3 + 3X^2 + 3X + 1 & n &:= \deg a = 5\,, \\ b(X) &:= 4X^3 + X^2 + X + 1 & m &:= \deg b = 3\,. \end{aligned}$$

Dazu normieren wir zunächst b zu

$$\bar{b}(X) = \tfrac{1}{4} b(X) = 4b(X) = X^3 + 4X^2 + 4X + 4$$

und berechnen

$$f(X) := b_m^*(X) = b_3^*(X) = 4X^3 + 4X^2 + 4X + 1\,.$$

Nun gilt also $f(0) = 1$ und wir können mit dem eigentlichen Invertieren $\mod X^l$ für $l = n - m + 1 = 3$, also Algorithmus 2.5, beginnen:
Setze $g_0 := 1$, $r := \lceil \log_2(3) \rceil = 2$.

$$\begin{aligned} i = 1: \quad g_1 &:= 2g_0 - fg_0^2 && \bmod X^{2^i} \\ &= 2 - (4X^3 + 4X^2 + 4X + 1) && \bmod X^2 \\ &= -4X + 1 = X + 1 \end{aligned}$$

Man beachte, dass der Term $2g_0 - fg_0^2$ lediglich für die Koeffizienten der Monome mit X^k für $k = 1$ interessant ist (vgl wieder Bemerkung 2.10)! Wie man in der vorliegenden Implementierung erkennt, wurde genau dies ausgenutzt und die Rechnung sieht in diesem Fall wie folgt aus:

$$\begin{aligned} i = 1: \quad g_1' &:= \texttt{multPInter}\ 2\ 1\ f\ (\texttt{multPInter}\ 2\ 0\ g_0\ g_0) \\ &= \texttt{multPInter}\ 2\ 1\ f\ 1 \\ &= 4X \\ g_1 &:= g_1' + g_0 = X + 1 \end{aligned}$$

Analog dazu sind im nächsten Schritt nur Terme mit X^k für $k = 3, 2$ interessant:

$$\begin{aligned} i = 2: \quad g'_2 \; &:= \texttt{multPInter}\ 4\ 2\ f\ (\texttt{multPInter}\ 4\ 0\ g_1\ g_1) \\ &= \texttt{multPInter}\ 4\ 2\ f\ (X^2 + X + 1) \\ &= 4X^3 + 2X^2 \\ g_2 \; &:= g'_2 + g_1 = 4X^3 + 2X^2 + X + 1 \end{aligned}$$

Das selbe Ergebnis erreicht man durch $g_2 := (2g_1 - fg_1^2) \bmod X^4$. Da $r = 2$, ist dies $g(X)$ mit $g(X)f(X) \equiv 1 \bmod X^3$. Nun können wir fortfahren mit Schritt 4 in Algorithmus 2.5 und

$$\begin{aligned} q'(X) &:= g(X)a_n^*(X) \bmod X^l \\ &= (4X^3 + 2X^2 + X + 1)(X^5 + 3X^4 + 3X^3 + 3X^2 + 4X + 1) \bmod X^3 \\ &= 4X^2 + 1 \end{aligned}$$

und damit letztlich

$$\begin{aligned} q(X) &:= b_m\ q'^*_{n-m}(X) = 4\ (4X^2 + 1)_2^* \\ &= 4(X^2 + 4) = 4X^2 + 1 \end{aligned}$$

berechnen. Den Rest erhalten wir dann durch

$$r(X) := q(X)b(X) - a(X) = 3X^2 + 2X\,.$$

2.3 Endliche Körper

2.3.1 Primkörper

Die Primkörper werden in dem Modul `GalFld.Core.PrimeFields` spezifiziert. Diese werden implementiert als `Int` Werte, die durch den Wrapper `Mod` noch zusätzlich die Information enthalten, in welchem Primkörper sich das Element befindet.

Da wir die Charakteristik zu einem solchem Körper auf Typebene speichern wollen, führen wir zunächst eine neue Typklasse mit dem Namen `Numeral` ein, welche als einzige Funktion `numValue :: a -> Int` besitzen. Diese Funktion soll konstant die Charakteristik wiedergeben.

Nun können wir durch

```
62 newtype Mod n = MkMod Int
```

GalFld/Core/PrimeFields.hs

Primkörper definieren, wobei für den Parameter `n` ein Datentyp von der Klasse `Numeral` eingesetzt werden soll.

Um zu einem Element im Primkörper einen Repräsentanten in $\mathbb{Z}$ zu erhalten, gibt es die Funktion

```
{-# INLINE unMod #-}
unMod :: Mod n → Int
unMod (MkMod k) = k
```

GalFld/Core/PrimeFields.hs

Einen Repräsentanten, der nichtnegativ, aber kleiner als die Charakteristik ist, liefert

```
{-# INLINE getRepr #-}
getRepr :: (Numeral n) ⇒ Mod n → Int
getRepr x = unMod x `mod` modulus x
```

GalFld/Core/PrimeFields.hs

Die Instanzen von **Show** und **ShowTex** ermöglichen es, Elemente von Primkörpern als String oder als LaTeX-Code darzustellen.

```
instance (Numeral n, Show n) ⇒ Show (Mod n) where
  show x = "\x1B[33m" ⧺ show (getRepr x) ⧺ "\x1B[39m" ⧺ showModulus x
    where showModulus :: (Numeral n) ⇒ Mod n → String
          showModulus = showModulus' . show . modulus
          showModulus' :: String → String
#if 1
          showModulus' "" = ""
          showModulus' (c:cs) = newC : showModulus' cs
            where newC | c ≡ '0' = '₀'
                       | c ≡ '1' = '₁'
                       | c ≡ '2' = '₂'
                       | c ≡ '3' = '₃'
                       | c ≡ '4' = '₄'
                       | c ≡ '5' = '₅'
                       | c ≡ '6' = '₆'
                       | c ≡ '7' = '₇'
                       | c ≡ '8' = '₈'
                       | c ≡ '9' = '₉'
#else
          showModulus' s = "^{" ⧺ s ⧺ "}"
#endif
```

GalFld/Core/PrimeFields.hs

```
instance (Numeral n, Show n) ⇒ ShowTex (Mod n) where
  showTex x = show (unMod x) ⧺ "_{" ⧺ show (modulus x) ⧺ "}"
```

GalFld/Core/PrimeFields.hs

Ferner implementieren wir Instanzen von **Eq**, **Num** und FiniteField.

```
instance (Numeral n) ⇒ Eq (Mod n) where
  {-# INLINE (==) #-}
  x ≡ y = (unMod x - unMod y) `rem` modulus x ≡ 0
```

GalFld/Core/PrimeFields.hs

```
instance (Numeral n) ⇒ Num (Mod n) where
  {-# INLINE (+) #-}
  x + y       = MkMod $ unMod x + unMod y `rem` modulus x
  {-# INLINE (*) #-}
  x * y       = MkMod $ unMod x * unMod y `rem` modulus x
```

GalFld/Core/PrimeFields.hs

```
  fromInteger = MkMod . fromIntegral
  abs x       = error "Prelude.Num.abs: inappropriate abstraction"
  signum _    = error "Prelude.Num.signum: inappropriate abstraction"
  {-# INLINE negate #-}
  negate      = MkMod . negate . unMod
```

```
instance (Numeral n) ⇒ FiniteField (Mod n) where
  zero            = MkMod 0
  one             = MkMod 1
  elems           = const $ elems' one
    where elems' :: (Numeral n) ⇒ Mod n → [Mod n]
          elems' x = map MkMod [0.. (modulus x - 1)]
  charakteristik = modulus
  elemCount      = modulus
  getReprP e     = 0 * snd (head (p2Tup e))
```

GalFld/Core/PrimeFields.hs

```
{-# INLINE modulus #-}
modulus :: Numeral a ⇒ Mod a → Int
modulus x = numValue $ modulus' x
  where modulus' :: Numeral a ⇒ Mod a → a
        modulus' = const undefined
```

GalFld/Core/PrimeFields.hs

Zum bequemen Invertieren haben wir auch noch eine Instanz von `Fractional` hinzugefügt.

```
instance (Numeral n) ⇒ Fractional (Mod n) where
  recip          = invMod
  fromRational _ = error "Prelude.Fractional.fromRational: inappropriate abstraction"
```

GalFld/Core/PrimeFields.hs

Weiterhin haben wir noch die folgenden Instanzen:

```
-- Zur Serialisierung wird eine Instanz vom Typ Binary benötigt
instance (Numeral a) ⇒ Binary (Mod a) where
  put (MkMod x) = put x
  get           = do x ← get
                     return $ MkMod x
```

GalFld/Core/PrimeFields.hs

```
instance (Numeral a, NFData a) ⇒ NFData (Mod a) where
  rnf = rnf . unMod
```

GalFld/Core/PrimeFields.hs

Erzeugen von Primkörpern

Möchte man nun einen Primkörper von beliebiger Charakteristik in einem Haskell- Programm benutzen, bietet sich die `TemplateHaskell` Funktion `genPrimeField` an. Diese übernimmt das Erzeugen von diversen Instanzen, die nötig sind.

GalFld/Core/PrimeFields.hs

```haskell
-- |Erzeugen von Primkörpern durch TemplateHaskell
genPrimeField :: Integer → String → Q [Dec]
genPrimeField p pfName = do
  d ← dataD (cxt []) (mkName mName) [] [] []
  i1 ← instanceD (cxt [])
    (appT (conT ''Numeral) (conT (mkName mName)))
    [funD (mkName "numValue")
      [clause [varP $ mkName "x"]
        (normalB $ litE $ IntegerL p) [] ] ]
  i2 ← instanceD (cxt [])
    (appT (conT ''Show) (conT (mkName mName)))
    [funD (mkName "show")
      [clause [] ( normalB $ appsE [varE (mkName "show")] ) [] ] ]
  i3 ← instanceD (cxt [])
    (appT (conT ''NFData) (conT (mkName mName))) []
  t ← tySynD (mkName pfName) []
    (appT (conT ''Mod) (conT $ mkName mName))
  return [d,i1,i2,t]
    where mName ='M' : show p
-- ppQ x = putStrLn =<< runQ ((show . ppr) `fmap` x)
```

Da es sich hierbei um eine Funktion handelt, die per `TemplateHaskell` zur Compilezeit ausgeführt wird, sind die beiden folgenden Pragmas nötig:

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
```

Dann kann beispielsweise der Primkörper der Charakteristik 7 namens `PF` durch

```haskell
$(genPrimeField 7 "PF")
```

erzeugt werden.

2.3.2 Erweiterungskörper

Um Erweiterungskörper darzustellen, verwenden wir Polynome, welche modulo einem Minimalpolynom gelesen werden sollen; codiert in dem Datentyp `FFElem`.

GalFld/Core/FiniteFields.hs

```haskell
-- Ein Element im Körper ist repräsentiert durch ein Paar von Polynomen. Das
-- erste beschreibt das Element, das zweite das Minimalpolynom
-- und damit den Erweiterungskörper.
-- Zusätzlich ist auch die kanonische Inklusion aus dem Grundkörper durch
-- FFKonst implementiert.
data FFElem a = FFElem (Polynom a) (Polynom a) | FFKonst a
```

Durch dieses Konzept kann man einfach in Erweiterungen von Erweiterungen rechnen. Startet man mit einem Primkörper, beispielsweise $\mathbb{F}_2$, so haben wir darin das Element 1:

```
f2 = 1::F2
```

Durch das Minimalpolynom $X^2 + X + 1$ erzeugen wir uns eine Erweiterung vom Grad 2:

```
e2f2Mipo = pList [1::F2,1,1] -- x^2+x+1
e2f2 = FFElem (pList [0,1::F2]) e2f2Mipo
```

Hier ist `e2f2` ein erzeugendes Element. Dies reicht, um alle Körperelemente generieren zu können. Durch eine weitere Grad 2 Erweiterung erhalten wir:

```
e2e2f2Mipo = pList [e2f2,one,one] -- x^2+x+e2f2
e2e2f2 = FFElem (pList [0,e2f2]) e2e2f2Mipo
```

Alternativ kann man auch durch eine Grad 4 Erweiterung über $\mathbb{F}_2$ den gleichen Körper erhalten:

```
e4f2Mipo = pList [1::F2,1::F2,0,0,1::F2] -- x^4+x^2+1
e4f2 = FFElem (pList [0,1::F2]) e4f2Mipo
```

Öffnet man `GalFld.Sandbox.FFSandbox` mit `GHCI` startet der Interpreter und man befindet sich in einer Umgebung, in der die Körper bereits erzeugt wurden. Nachdem wir also bereits das Element `e2e2f2` haben, können wir uns dieses anzeigen lassen, indem wir einfach nur `e2e2f2` in die Konsole eintippen und bestätigen. Damit erhalten wir

```
((1₂·X mod 1₂·X²+1₂·X+1₂)·X mod (1₂ mod ...)·X²+(1₂ mod ...)·X+(1₂·X mod 1₂·X²+1₂·X+1₂))
```

Ein Element in einer Körpererweiterung wird beispielsweise dargestellt als

- `(1₂·X mod 1₂·X²+1₂·X+1₂)`, welches die Äquivalenzklasse von x in $\mathbb{F}_2[X]/(X^2 + X + 1)$ bezeichnet. Die LaTeX Darstellung dazu ist $\left(\underline{1_2 \cdot X}_{mod\ 1_2 \cdot X^2 + 1_2 \cdot X + 1_2}\right)$.
- `(1₂ mod ...)` bedeutet, dass noch nicht klar ist, modulo welchem Polynom dieses Element gelesen wird. Es ist also die $1 \in \mathbb{F}_2[x]/(g(x))$ wobei $g(x)$ erst während der Berechnung inferiert werden muss. Dies ist nötig, um die Inklusion des Grundkörpers zu realisieren.

Durch die `ShowTex`-Instanz können wir `e2e2f2` auch in LaTeX darstellen:

$$\left(\underline{\left(\underline{1_2 \cdot X}_{mod\ 1_2 \cdot X^2 + 1_2 \cdot X + 1_2}\right) \cdot X}_{mod\ 1_2 \cdot X^2 + 1_2 \cdot X + \left(\underline{1_2 \cdot X}_{mod\ 1_2 \cdot X^2 + 1_2 \cdot X + 1_2}\right)}\right)$$

Ersetzen wir `(1₂·X mod 1₂·X²+1₂·X+1₂)` mit `Y` dann kann `e2e2f2` auch geschrieben werden als:

```
(Y·X mod (1₂ mod ...)·X²+(1₂ mod ...)·X+Y)
```

Dieses Element ist also die Äquivalenzklasse von YX in $\mathbb{F}_2[Y,X]/(Y^2+Y+1, X^2+X+Y)$.

Nun können wir z.B. `e2e2f2 + e2e2f2 * e2e2f2` berechnen und erhalten:

$$((1_2 \ \mathtt{mod}\ 1_2\cdot X^2+1_2\cdot X+1_2)\cdot X+(1_2 \ \mathtt{mod}\ 1_2\cdot X^2+1_2\cdot X+1_2) \ \mathtt{mod}\ (1_2 \ \mathtt{mod}\ \ldots)\cdot X^2+(1_2 \ \mathtt{mod}\ \ldots)\cdot X+(1_2\cdot X \ \mathtt{mod}\ 1_2\cdot X^2+1_2\cdot X+1_2))$$

In LaTeX:

$$\left(\underline{\left(\underline{1_2}_{mod\ 1_2\cdot X^2+1_2\cdot X+1_2}\right)\cdot X+\left(\underline{-1_2}_{mod\ 1_2\cdot X^2+1_2\cdot X+1_2}\right)}_{mod\ 1_2\cdot X^2+1_2\cdot X+}\left(\underline{1_2\cdot X}_{mod\ 1_2\cdot X^2+1_2\cdot X+1_2}\right)\right)$$

Funktionen auf Körpererweiterungen

Wie wir gesehen haben, ist der zugrunde liegende Körper nicht bei jedem Koeffizienten eines Polynoms klar. Daher liefert `getReprP` für ein Polynom einen passenden Repräsentanten und `charOfP` als abkürzende Schreibweise dessen Charakteristik.

```haskell
{-# INLINE getReprP' #-}
getReprP' f = getReprP'' $ p2Tup f
getReprP'' []                   =
                                  error "Insufficient information in this Polynomial"
getReprP'' ((i,FFKonst _):ms)   = getReprP'' ms
getReprP'' ((i,FFElem f p): ms) = FFElem 0 p
```

GalFld/Core/FiniteFields.hs

```haskell
{-# INLINE charOfP #-}
-- |Gibt die Charakteristik der Koeffizienten eines Polynoms
charOfP :: (Eq a, FiniteField a, Num a) => Polynom a -> Int
charOfP f = charakteristik $ getReprP f
```

GalFld/Core/FiniteFields.hs

Bekanntlich ist $(\_)^p$ auf endlichen Körpern der Charakteristik p ein Automorphismus, was das Ziehen von p-ten Wurzeln rechtfertigt. Sicherlich gibt es dazu bessere Algorithmen, jedoch haben wir uns aufgrund der Einfachheit entschieden, dies in $\mathbb{F}_{p^m}$ durch $(\_)^{p^{m-1}}$ zu implementieren.

```haskell
{-# INLINE charRootP #-}
-- |Zieht die p-te Wurzel aus einem Polynom, wobei p die Charakteristik ist
charRootP :: (Show a, FiniteField a, Num a) => Polynom a -> Polynom a
charRootP f | isNullP f      = nullP
            | f == pKonst 1  = pKonst 1
            | otherwise      = pTupUnsave [(i `quot` p,m^l) | (i,m) <- p2Tup f]
  where p = charOfP f
        q = elemCount $ getReprP f
        l = max (quot q p) 1
```

GalFld/Core/FiniteFields.hs

Instanzen

Die implementierten Instanzen sind selbstredend gleich denen der Primkörper.

```haskell
instance (Show a, Num a, Eq a, Fractional a) ⇒ Eq (FFElem a) where
  (FFKonst x)  ≡ (FFKonst y)  = x≡y
  (FFElem f p) ≡ (FFKonst y)  = isNullP $ f - pKonst y
  (FFKonst x)  ≡ (FFElem g p) = isNullP $ g - pKonst x
  (FFElem f p) ≡ (FFElem g q) | p≡q        = isNullP $ f-g
                              | otherwise = error "Not the same mod"
```

GalFld/Core/FiniteFields.hs

```haskell
instance (Show a, Num a, Eq a) ⇒ Show (FFElem a) where
  show (FFKonst x)                  = "(" ⧺ show x ⧺ " mod ...)"
  show (FFElem f p) | isNullP f     = "(0 mod " ⧺ show p ⧺ ")"
                    | otherwise     = "(" ⧺ show f ⧺ " mod " ⧺ show p ⧺")"
```

GalFld/Core/FiniteFields.hs

```haskell
instance (ShowTex a, Num a, Eq a) ⇒ ShowTex (FFElem a) where
  showTex (FFKonst x) = showTex x
  showTex (FFElem f p)
    | isNullP f    = "\\left(\\underline{0}_{mod~" ⧺ showTex p ⧺ "}\\right)"
    | otherwise    =
      "\\left(\\underline{" ⧺ showTex f ⧺ "}_{mod~" ⧺ showTex p ⧺"}\\right)"
```

GalFld/Core/FiniteFields.hs

```haskell
instance (Show a, Num a, Eq a, Fractional a) ⇒ Num (FFElem a) where
  fromInteger i                           = FFKonst (fromInteger i)
  {-# INLINE (+) #-}
  (FFKonst x)  + (FFKonst y)              = FFKonst (x+y)
  (FFElem f p) + (FFKonst x)              = FFElem (f + pKonst x) p
  (FFKonst x)  + (FFElem f p)             = FFElem (f + pKonst x) p
  (FFElem f p) + (FFElem g q) | p≡q       = aggF $ FFElem (f+g) p
                              | otherwise = error "Not the same mod"
  {-# INLINE (*) #-}
  (FFKonst x)  ∗ (FFKonst y)              = FFKonst (x*y)
  (FFElem f p) ∗ (FFKonst x)              = FFElem (f ∗ pKonst x) p
  (FFKonst x)  ∗ (FFElem f p)             = FFElem (f ∗ pKonst x) p
  (FFElem f p) ∗ (FFElem g q) | p≡q       = aggF $ FFElem (f*g) p
                              | otherwise = error "Not the same mod"
  {-# INLINE negate #-}
  negate (FFKonst x)                      = FFKonst (negate x)
  negate (FFElem f p)                     = FFElem (negate f) p
  abs _    = error "Prelude.Num.abs: inappropriate abstraction"
  signum _ = error "Prelude.Num.signum: inappropriate abstraction"
```

GalFld/Core/FiniteFields.hs

```haskell
instance (Show a, Eq a, Fractional a) ⇒ Fractional (FFElem a) where
  fromRational _     = error "inappropriate abstraction"
  {-# INLINE recip #-}
  recip (FFKonst x)  = FFKonst (recip x)
  recip (FFElem f p) | isNullP f     = error "Division by zero"
                     | otherwise     = FFElem s p
    where (_,s,_) = eekP f p
```

GalFld/Core/FiniteFields.hs

```
instance (Num a, Eq a, NFData a) ⇒ NFData (FFElem a) where
  rnf (FFElem f p) = rnf (f,p)
  rnf (FFKonst x)  = rnf x
```

GalFld/Core/FiniteFields.hs

```
instance (Num a, Binary a) ⇒ Binary (FFElem a) where
  put (FFKonst f)  = do put (0 :: Word8)
                        put f
  put (FFElem f p) = do put (1 :: Word8)
                        put f
                        put p
```

GalFld/Core/FiniteFields.hs

2.4 Lineare Algebra

GalFld/Core/Matrix.hs

Grundlegende Funktionen der linearen Algebra – wie man sie im weiteren Verlauf beispielsweise für den Berlekamp-Algorithmus brauchen wird – haben wir in der Datei Core/Matrix.hs hinterlegt.

Eine Matrix ist dabei der folgende Datentyp:

```
-- Eine Matrix ist als zweidimensionales Array dargestellt,
-- wobei die erste Stelle die Zeile und dei zweite die Spalte entspricht.
data Matrix a = M {unM :: Array (Int, Int) a} | Mdiag a
```

GalFld/Core/Matrix.hs

Es hätte auch die Möglichkeit bestanden, Matrizen als [[a]] (also als doppelte Liste) zu implementieren, jedoch haben Listen eine Zugriffszeit von $\mathcal{O}(l)$ auf das l-te Element und die Abfrage der Länge dauert bei einer Liste der Länge n $\mathcal{O}(n)$. Arrays schaffen beides in $\mathcal{O}(1)$, jedoch mit einer weit größeren Konstante (vgl.).

2.4.1 Erzeugung von Matrizen und Basisoperationen

Erzeugung Entweder erzeugt man eine Matrix direkt als Array (Int,Int) a oder durch die Verwendung von fromListsM.

```
{-# INLINE fromListsM #-}
-- |Erzeugt eine Matrix aus einer Liste von Listen von Einträgen
fromListsM :: [[a]] → Matrix a
fromListsM []   = error "GalFld.Core.Matrix.fromListsM: empty lists"
fromListsM [[]]  = error "GalFld.Core.Matrix.fromListsM: empty lists"
fromListsM ess = M $ array ((1,1),(k,l))
                           [((i,j),ess!!(i-1)!!(j-1)) | i ← [1..k]
                                                      , j ← [1..l]]
  where k = length ess
        l = length $ head ess
```

GalFld/Core/Matrix.hs

Für den Spezialfall des Vielfachen der Einheitsmatrix kann man auch folgende Funktion verwenden.

GalFld/Core/Matrix.hs

```haskell
{-# INLINE genDiagM #-}
-- |Erzeugt ein Vielfaches der Einheitsmatrix
genDiagM :: Num a => a -> Int -> Matrix a
genDiagM x n = M $ array ((1,1),(n,n)) $ fillList [((i,i),x) | i <- [1..n]] n n
  where fillList ls n m = ls ++ [(idx,0) | idx <- getAllIdxsExcept n m idxs]
          where idxs                        = map fst ls
                getAllIdxsExcept n m idxs = [idx | idx <- [(i,j) | i <- [1..n]
                                                                  , j <- [1..m]]
                                                  , idx `notElem` idxs]
```

Beispiel 2.13 Möchte man die Matrix $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \in \mathbb{Z}^{2\times 2}$ erzeugen, so gibt es drei verschiedene Varianten:

1. `array ((1,1),(2,2)) [((1,1),2::Int), ((1,2),2::Int), ((2,1),2::Int), ((2,2),2::Int)]`
2. `genDiagM (2::Int) 2`
3. `fromListsM [[2::Int,0],[0,2]]`

Bemerkung 2.14 Es gilt anzumerken, dass der Konstruktor für `Array` stets eine *vollständige* Liste erwartet. (Vergleiche auch die interne Funktion `getAllIdxsExcept` in `genDiagM`.)

Basisoperationen Selbstredend möchte man eine Matrix auch wieder in Listenform zurückverwandeln:

GalFld/Core/Matrix.hs

```haskell
{-# INLINE toListsM #-}
-- |Erzeugt aus einer Matrix eine Liste von Listen der Einträge. Ist invers zu
-- fromListsM
toListsM :: Matrix a -> [[a]]
toListsM (M m) = [[m!(i,j) | j <- [1..l]] | i <- [1..k]]
  where (k,l) = snd $ bounds m
```

Die Dimension und Anzahl der Spalten bzw. Zeilen einer Matrix lässt sich durch die Arraydarstellung sehr leicht angeben.

GalFld/Core/Matrix.hs

```haskell
-- |Gibt zu einer Matrix die Grenzen zurück
-- Das Ergebnis hat die Form ((1,k),(1,l))
{-# INLINE boundsM #-}
boundsM :: Matrix a -> (Int,Int)
boundsM (M m) = snd $ bounds m
```

GalFld/Core/Matrix.hs

```haskell
-- |Gibt zu einer Matrix die Anzahl der Zeilen zurück
{-# INLINE getNumRowsM #-}
getNumRowsM :: Matrix a -> Int
getNumRowsM (M m) = fst $ snd $ bounds m
```

GalFld/Core/Matrix.hs

```
-- |Gibt zu einer Matrix die Anzahl der Spalten zurück
{-# INLINE getNumColsM #-}
getNumColsM :: Matrix a → Int
getNumColsM (M m) = snd $ snd $ bounds m
```

Ebenfalls sehr leicht ist ein Test, ob eine quadratische Matrix vorliegt.

GalFld/Core/Matrix.hs

```
{-# INLINE isQuadraticM #-}
isQuadraticM :: Matrix a → Bool
isQuadraticM (Mdiag a) = True
isQuadraticM (M m) = uncurry (≡) $ snd $ bounds m
```

Ein Element einer Matrix an einer bestimmten Stelle findet man wie folgt.

GalFld/Core/Matrix.hs

```
-- |Gibt zu einer Matrix den Wert an der Position (row,col) zurück
{-# INLINE atM #-}
atM :: Matrix a → Int → Int → a
atM (M m) row col = m!(row,col)
```

Eine ganze Zeile bzw. Spalte bekommt man durch `getRowM` bzw. `getColM`.

GalFld/Core/Matrix.hs

```
-- |Gibt zu einer Matrix die i-te Zeile zurück
{-# INLINE getRowM #-}
getRowM :: Matrix a → Int → [a]
getRowM (M m) i = [m!(i,j) | j ← [1..l]]
  where (k,l) = snd $ bounds m
```

GalFld/Core/Matrix.hs

```
-- |Gibt zu einer Matrix die i-te Spalte zurück
{-# INLINE getColM #-}
getColM :: Matrix a → Int → [a]
getColM (M m) i = [m!(j,i) | j ← [1..k]]
  where (k,l) = snd $ bounds m
```

Aneinanderfügen von Matrizen Wenn man zwei Matrizen horizontal bzw. vertikal aneinanderfügt, erhält man eine neue Matrix. Dies ist gerade beim Anwenden des Gaußschen Eliminationsverfahrens von großem Nutzen.

GalFld/Core/Matrix.hs

```
{-# INLINE (<|>) #-}
-- |Horizontales Aneinanderfügen von Matrizen
(<|>) :: Matrix a → Matrix a → Matrix a
(<|>) (M m1) (M m2) =  M $ array ((1,1),(k1,l1+l2)) $ assocs m1 ++
                             assocs (ixmap ((1,l1+1),(k2,l1+l2))
                                    (λ(i,j) → (i,j-l1)) m2)
  where (k1,l1) = snd $ bounds m1
        (k2,l2) = snd $ bounds m2
{-# INLINE (<->) #-}
-- |Vertikales Aneinanderfügen von Matrizen
(<->) :: Matrix a → Matrix a → Matrix a
(<->) (M m1) (M m2) =  M $ array ((1,1),(k1+k2,l1)) $ assocs m1 ++
```

```
                                    assocs (ixmap ((k1+1,1),(k1+k2,l1))
                                           (\(i,j) -> (i-k1,j)) m2)
  where (k1,l1) = snd $ bounds m1
        (k2,l2) = snd $ bounds m2
```

GalFld/Core/Matrix.hs

```
{-# INLINE (<->) #-}
-- |Vertikales Aneinanderfügen von Matrizen
(<->) :: Matrix a -> Matrix a -> Matrix a
(<->) (M m1) (M m2) =  M $ array ((1,1),(k1+k2,l1)) $ assocs m1 ++
                                    assocs (ixmap ((k1+1,1),(k1+k2,l1))
                                           (\(i,j) -> (i-k1,j)) m2)
  where (k1,l1) = snd $ bounds m1
        (k2,l2) = snd $ bounds m2
```

Untermatrizen Untermatrizen erhält man wie folgt.

GalFld/Core/Matrix.hs

```
{-# INLINE subM #-}
-- |Gibt zu einer Matrix eine Untermatrix zurück
-- Input:
--        (k0,l0) : erste übernommene Spalte und Zeile
--        (k1,l1) : letzte übernommene Spalte und Zeile
--        m         : Eingabematrix
subM :: Num a => (Int,Int) -> (Int,Int) -> Matrix a -> Matrix a
subM (k0,l0) (k1,l1) (Mdiag x) = subM (k0,l0) (k1,l1) $ genDiagM x $ max k1 l1
subM (k0,l0) (k1,l1) (M m)     = M $ subArr (k0,l0) (k1,l1) m
```

GalFld/Core/Matrix.hs

```
{-# INLINE subArr #-}
-- |Gibt zu einer Matrix eine Untermatrix zurück
subArr :: Num a => (Int,Int) -> (Int,Int) -> Array (Int, Int) a
                                                  -> Array (Int, Int) a
subArr (k0,l0) (k1,l1) m =  array ((1,1),(k,l))
    [ ((i-k0+1,j-l0+1) , m!(i,j)) | i <- [k0..k1] , j <- [l0..l1]]
  where !(k,l) = (k1-k0+1,l1-l0+1)
```

Vertauschen von Zeilen bzw. Spalten Ebenfalls beim Gauß-Verfahren vonnöten ist das Vertauschen von Zeilen bzw. Spalten.

GalFld/Core/Matrix.hs

```
{-# INLINE swapRowsM #-}
-- |Vertauscht zwei Zeilen in einer Matrix
swapRowsM :: Num a => Int -> Int -> Matrix a -> Matrix a
swapRowsM _ _ (Mdiag x) =
  error "GalFld.Core.Matrix.swapRowsM: Not enough information given"
swapRowsM k0 k1 (M m)    = M $ swapRowsArr k0 k1 m
```

GalFld/Core/Matrix.hs

```
{-# INLINE swapColsM #-}
-- |Vertauscht zwei Spalten in einer Matrix
swapColsM :: Num a => Int -> Int -> Matrix a -> Matrix a
swapColsM _ _ (Mdiag x) =
```

```
  error "GalFld.Core.Matrix.swapColsM: Not enough information given"
swapColsM l0 l1 (M m) = M $ swapColsArr l0 l1 m
```

GalFld/Core/Matrix.hs

```
{-# INLINE swapRowsArr #-}
-- |Vertauscht zwei Zeilen in einem Array, das zu einer Matrix gehört
swapRowsArr :: Num a ⇒ Int → Int → Array (Int, Int) a → Array (Int, Int) a
swapRowsArr k0 k1 m = array ((1,1),(k,l))
    [ ((swp i,j) , m!(i,j)) | i ← [1..k] , j ← [1..l]]
  where (k,l) = snd $ bounds m
        swp i | i ≡ k0     = k1
              | i ≡ k1     = k0
              | otherwise = i
```

GalFld/Core/Matrix.hs

```
{-# INLINE swapColsArr #-}
-- |Vertauscht zwei Spalten in einem Array, das zu einer Matrix gehört
swapColsArr :: Num a ⇒ Int → Int → Array (Int, Int) a → Array (Int, Int) a
swapColsArr l0 l1 m = array ((1,1),(k,l))
    [ ((i,swp j) , m!(i,j)) | i ← [1..k] , j ← [1..l]]
  where (k,l) = snd $ bounds m
        swp j | j ≡ l0     = l1
              | j ≡ l1     = l0
              | otherwise = j
```

2.4.2 Zweiwertige Operationen auf Matrizen

Addition Die Addition zweier Matrizen erklärt sich von selbst.

GalFld/Core/Matrix.hs

```
{-# INLINE addM #-}
addM :: (Num a) ⇒ Matrix a → Matrix a → Matrix a
addM (Mdiag x) (Mdiag y) = Mdiag (x+y)
addM (Mdiag x) m         = addM m (genDiagM x (getNumRowsM m))
addM m         (Mdiag y) = addM m (genDiagM y (getNumRowsM m))
addM (M x)     (M y)     | boundTest = M $ array (bounds x)
  [(idx,x!idx + y!idx) | idx ← indices x]
                         | otherwise =
  error "GalFld.Core.Matrix.addM: not the same Dimensions"
    where boundTest = bounds x ≡ bounds y
```

Multiplikation Die Multiplikation wurde nach dem Standardverfahren implementiert.

GalFld/Core/Matrix.hs

```
{-# INLINE multM #-}
multM :: (Num a) ⇒ Matrix a → Matrix a → Matrix a
multM (Mdiag x) (Mdiag y) = Mdiag (x*y)
multM (Mdiag x) m         = multM (genDiagM x (getNumRowsM m)) m
multM  m        (Mdiag x) = multM m (genDiagM x (getNumColsM m))
multM (M m)     (M n)     | k' ≡ l     = M $ array ((1,1),(k,l'))
```

```
  [((i,j), sum [m!(i,k) * n!(k,j) | k ← [1..l]]) | i ← [1..k] , j ← [1..l']]
                        | otherwise =
  error "GalFld.Core.Matrix.multM: not the same Dimensions"
    where ((_,_),(k,l))   = bounds m
          ((_,_),(k',l')) = bounds n
```

2.4.3 Lineare Algebra

Transponieren

GalFld/Core/Matrix.hs

```
{-# INLINE transposeM #-}
-- |Transponieren einer Matrix
transposeM :: Matrix a → Matrix a
transposeM (Mdiag a) = Mdiag a
transposeM (M m)     = M $ ixmap ((1,1),(l,k)) (λ(x,y) → (y,x)) m
  where !(k,l) = snd $ bounds m
```

Zeilenstufenform Um eine Matrix in Zeilenstufenform zu bringen, verwenden wir den allseits bekannten Algorithmus.

GalFld/Core/Matrix.hs

```
-- |Berechnet die Zeilenstufenform einer Matrix
{-# INLINE echelonM #-}
echelonM :: (Show a, Eq a, Num a, Fractional a) ⇒ Matrix a → Matrix a
echelonM (Mdiag n) = Mdiag n
echelonM (M m)     = M $ echelonM' m
  where echelonM' :: (Show a, Eq a, Num a, Fractional a) ⇒
                     Array (Int,Int) a → Array (Int,Int) a
        echelonM' m | k ≡ 1         = arrElim m
                    | l ≡ 1         = arrElim m
                    | hasPivot      = echelonM' $ swapRowsArr 1 (minimum lst) m
                    | noPivot       = echelonM'_noPivot m
                    | otherwise     = echelonM'_Pivot m
          where !(k,l)     = snd $ bounds m
                !lst       = [i | i ← [1..k], m!(i,1) ≠ 0]
                !hasPivot  = m!(1,1) ≡ 0 && not (null lst)
                !noPivot   = m!(1,1) ≡ 0 && null lst
```

Nahezu selbsterklärend beginnt der Algorithmus mit einer Fallunterscheidung. Ist das aktuell zu bearbeitende Element 0 und die gesamte darunterliegende Spalte auch, so geht es mit `echelonM'_noPivot` weiter. Ist der aktuelle Eintrag 0, wird aber ein Pivotelement gefunden, so vertauscht man die Zeilen passend (`swapRowsArr`). Ist der aktuelle Eintrag $\neq 0$, so liefert `echelonM'_Pivot` den passenden Eliminationsschritt durch die Anwendung von `arrElim`.

GalFld/Core/Matrix.hs

```
{-# INLINE arrElim #-}
-- |Zieht die erste Zeile passend von allen anderen ab, eliminiert also in
-- jeder außer der ersten Zeile den ersten Eintrag der Zeile
```

```haskell
arrElim :: (Eq a, Num a, Fractional a) => Array (Int, Int) a
                                               -> Array (Int, Int) a
arrElim m | m!(1,1) == 0 = m
          | otherwise    =
  (m // [ ((1,j),m!(1,j)/m!(1,1)) | j <- [1..l]])
    // [ ((i,j), m!(i,j) - m!(i,1) / m!(1,1) * m!(1,j)) | j <- [1..l],
        i <- [2..k]]
  where !(k,l) = snd $ bounds m
```

Beispiel 2.15 Wir versuchen einmal die „Numblock-Matrix“

$$M = \begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 4 \\ 4 & 0 & 1 \\ 1 & 2 & 3 \end{bmatrix} \in \mathbb{F}_5^{3\times 3}$$

in Zeilenstufenform zu bringen. Lässt man sich die einzelnen Zwischenschritte von `echelonM` ausgeben, so erhält man:

`echelonM' (k,l)=(3,3) m=`
$2_5\ 3_5\ 4_5$
$4_5\ 0_5\ 1_5$
$1_5\ 2_5\ 3_5$
`        ->(1,1)/=0`
`echeonM'_Pivot m'=`
$1_5\ 4_5\ 2_5$
$0_5\ 4_5\ 3_5$
$0_5\ 3_5\ 1_5$
`echelonM' (k,l)=(2,2) m=`
$4_5\ 3_5$
$3_5\ 1_5$
`        ->(1,1)/=0`
`echeonM'_Pivot m'=`
$1_5\ 2_5$
$0_5\ 0_5$

Die eigentliche Ausgabe der Funktion `echelonM` ist dann selbstverständlich:

$1_5\ 4_5\ 2_5$
$0_5\ 1_5\ 2_5$
$0_5\ 0_5\ 0_5$

Kern Mit Hilfe der Zeilenstufenform kann man wie folgt den Kern einer Matrix berechnen: Fügt man die Einheitsmatrix passender Größe unten an die ursprüngliche Matrix an und berechnet dann die Spaltenstufenform der zusammengesetzten Gesamtmatrix, so sind die Nichtnullspalten des unteren Teils des Ergebnisses gerade eine Basis des Kerns der ursprünglichen Matrix (vgl. [19, Abschnitt Basis]). Dies lässt sich durch Transponieren natürlich leicht auf die Berechnung einer Zeilenstufenform zurückführen.

GalFld/Core/Matrix.hs

```
-- |Berechnet den Kern einer Matrix, d.h.
--  kernelM gibt eine Matrix zurück, deren Spalten eine Basis des
--  Kerns sind
{-# INLINE kernelM #-}
kernelM :: (Show a, Eq a, Num a, Fractional a) ⇒ Matrix a → Matrix a
kernelM (Mdiag m) = error "GalFld.Core.Matrix.kernelM: No kernel here"
kernelM m      = M $ array ((1,1), (k,lzs))
                  [ ((i,j),b!(i,zs!!(j-1))) | i ← [1..k], j ← [1..lzs]]
  where !(k,l) = snd $ bounds $ unM m
        !mfull = transposeM $ echelonM $
                 transposeM $ m <-> genDiagM 1 k
        !a     = subArr (1,1) (k,l) $ unM mfull
        !b     = subArr (k+1,1) (k+k,l) $ unM mfull
        !zs    = [j | j ← [1..l], and [a!(i',j) ≡ 0 | i' ← [j..k]]]
        !lzs   = length zs
```

Determinante Offensichtlich lässt sich in Zeilenstufenform auch die Determinante einer Matrix berechnen.

GalFld/Core/Matrix.hs

```
{-# INLINE detM #-}
-- |Berechnet die Determinante effektiver als detLapM; braucht aber Fractional
detM :: (Eq a, Num a, Fractional a) ⇒ Matrix a → a
detM (Mdiag 0) = 0
detM (Mdiag 1) = 1
detM (Mdiag _) =
  error "GalFld.Core.Matrix.detM: Not enough information given"
detM m          | isQuadraticM m = detArr $ unM m
                | otherwise      =
  error "GalFld.Core.Matrix.detM: Matrix not quadratic"
    where {-# INLINE detArr #-}
          -- |detM auf Array-Ebene
          detArr :: (Eq a, Num a, Fractional a) ⇒ Array (Int, Int) a → a
          detArr m | k ≡ 1        = m!(1,1)
                   | m!(1,1) ≡ 0 = - detArrPivot m
                   | otherwise    = (m!(1,1) *) $ detArr $ subArr (2,2) (k,l) $
                                    arrElim m
            where !(k,l) = snd $ bounds m
```

Hier wurde der Algorithmus zur Zeilenstufenform nahezu erneut implementiert, um die konkrete Elimination in den einzelnen Spalten auslassen zu können, die bei der Berechnung der Determinante ja unnötig ist.

Determinante ohne `Fractional` Bekanntlich lässt sich die Determinante einer Matrix über jedem Ring definieren. Das bedeutet, dass es auch ohne die zur Berechnung der Zeilenstufenform nötigen `Fractional`-Instanz geht, was `detLapM` liefert.[5]

[5] Man hätte auch die Berechnung der Smithschen-Normalform implementieren können, die ebenfalls ohne `Fractional` ausgekommen wäre. Jedoch haben wir uns entschieden darauf zu verzichten, da

GalFld/Core/Matrix.hs

```haskell
{-# INLINE detLapM #-}
-- |Berechne die Determinante ohne Nutzen von Fractional a
detLapM :: (Eq a, Num a) => Matrix a -> a
detLapM (Mdiag 0) = 0
detLapM (Mdiag 1) = 1
detLapM (Mdiag _) =
  error "GalFld.Core.Matrix.detLapM: Not enough information given"
detLapM m | isQuadraticM m = detLapM' $ unM m
          | otherwise      = 0
{-# INLINE detLapM' #-}
detLapM' :: (Eq a, Num a) => Array (Int, Int) a -> a
detLapM' m | b == (1,1) = m!(1,1)
           | otherwise =
  sum [(-1)^(i-1) * m!(i,1) * detLapM' (getSubArr i) | i <- [1..fst b]]
    where !b           = snd $ bounds m
          {-# INLINE getSubArr #-}
          getSubArr i = array ((1,1),(fst b-1,snd b-1)) $
            [((i',j'),m!(i',j'+1)) | i' <- [1..(i-1)]
                                   , j' <- [1..(snd b - 1)]]
            ++ [((i',j'),m!(i'+1,j'+1)) | i' <- [i..(fst b - 1)]
                                        , j' <- [1..(snd b - 1)]]
```

2.4.4 Weiteres

Alle Matrizen mit vorgegebenen Einträgen Möchte man alle Matrizen erzeugen, die eine vorgegebene Größe und vorgegebene Einträge besitzen, so liefert `getAllM` die Antwort.

GalFld/Core/Matrix.hs

```haskell
{-# INLINE getAllM #-}
-- |Gibt eine Liste aller Matrizen, welche Einträge aus einer Liste besitzen
-- und eine gewisse Größe haben, zurück
getAllM :: [a] -> (Int,Int) -> [Matrix a]
getAllM cs (k,l) = map fromListsM $ rowMs k
  where lines = lines' l
        lines' n | n == 1      = [[y] | y <- cs]
                 | otherwise = [y:ys | y <- cs, ys <- lines' (n-1) ]
        rowMs n  | n == 1      = [[y] | y <- lines]
                 | otherwise = [y:ys | y <- lines, ys <- rowMs (n-1) ]
```

2.5 Faktorisierung von Polynomen über endlichen Körpern

GalFld/Core/Factorization.hs
GalFld/Algorithmen/Berlekamp.hs

im vorliegenden Anwendungsfall der endlichen Körper stets Inverse zur Verfügung stehen.

`GalFld/Algorithmen/Rabin.hs`

Faktorisierungsalgorithmen Grundsätzlich sind alle Algorithmen, die in Kapitel 3 beschrieben werden und der konkreten Faktorisierung von Polynomen dienen, als `Polynom a →[(Int, Polyno` beschrieben. Das bedeutet, dass eine *Faktorisierung* stets als Liste von Tupeln zu verstehen ist, wobei der erste Eintrag die Multiplizität des zweiten Eintrags, dem konkreten Faktor, angibt.

2.5.1 Triviale Faktoren

Kann man aus einem Polynom $f(X)$ den trivialen Faktor X ausklammern, so leistet dies die Funktion `obviousFactor`.

`GalFld/Core/Factorization.hs`

```haskell
obviousFactor :: (Show a, Num a, Eq a) ⇒ Polynom a → [(Int,Polynom a)]
obviousFactor f | isNullP f     = [(1,nullP)]
                | uDegP f ≤ 1   = [(1,f)]
                | hasNoKonst fs = factorX
                | otherwise     = toFact f
  where fs = p2Tup f
        -- Teste, ob ein konstanter Term vorhanden ist
        hasNoKonst ms | fst (last ms) ≡ 0  = False
                      | otherwise          = True
        -- Hier kann man d mal X ausklammern
        factorX | g ≡ 1     = [(d,pTupUnsave [(1,1)])]
                | otherwise = [(d,pTupUnsave [(1,1)]), (1,g)]
          where d = fst $ last fs
                g = pTupUnsave $ map (A.first (λx → x-d)) fs
```

Um Polynome, die keinen trivialen Faktor besitzen trotzdem als Faktorisierung zurückgegeben zu können, haben wir `toFact` implementiert.

`GalFld/Core/Factorization.hs`

```haskell
-- |Erzeugt eine triviale Faktorisierung zu einem Polynom
toFact :: Polynom a → [(Int,Polynom a)]
toFact f = [(1,f)]
```

2.5.2 Funktionen rund um Faktorisierungen

Anwenden von Faktorisierungen Es ist klar, dass man verschiedene Algorithmen kombinieren will, die teilweise Faktorisierungen herstellen (vgl. z.B. quadratfreie Faktorisierung mit anschließendem Berlekamp). Dazu braucht man Funktionen, die einen Faktorisierungsalgorithmus (also eine Funktion `Polynom a →[(Int,Polynom a)]`) auf eine bereits vorhandene Faktorisierung anwenden und anschließend das Ergebnis zusammenfassen. Hierfür gibt es den nachstehenden Wrapper `appFact`.

GalFld/Core/Factorization.hs

```
-- |Nimmt eine Faktorisierung und wendet auf diese einen gegebenen
-- Faktorisierungsalgorithmus an
appFact :: (Eq a, Num a) =>
  (Polynom a -> [(Int,Polynom a)]) -> [(Int,Polynom a)] -> [(Int,Polynom a)]
appFact alg = withStrategy (parList rpar) . concatMap
  (uncurry appFact')
  where appFact' i f  | isNullP f   = [(i,nullP)]
                      | uDegP f <= 1 = [(i,f)]
                      | otherwise   = potFact i (alg f)
```

`potFact` fasst dabei die entstehenden Mehrfachpotenzen der Faktoren zusammen.

GalFld/Core/Factorization.hs

```
-- |Ersetzt eine Faktorisierung, durch die n-te Potenz dieser Faktorisierung
potFact :: (Num a) => Int -> [(Int,Polynom a)] -> [(Int,Polynom a)]
potFact _ []          = []
potFact n ((i,f):ts) = (i*n,f) : potFact n ts
```

Es gilt zu bemerken, dass `appFact` parallelisiert ausgeführt wird, sofern die Multicore-Unterstützung aktiviert wurde.

Man will jedoch als Anwender nicht immer `appFact` auf einen Algorithmus anwenden. Daher gibt es für jeden Faktorisierungsalgorithmus eine Funktion app$A$ wobei A für den jeweiligen konkreten Algorithmus steht.

GalFld/Core/Factorization.hs

```
appObFact :: (Show a, Num a, Eq a) => [(Int,Polynom a)] -> [(Int,Polynom a)]
appObFact = appFact obviousFactor
```

GalFld/Algorithmen/SFreeFactorization.hs

```
appSff :: (Show a, FiniteField a, Num a, Fractional a) =>
                                        [(Int,Polynom a)] -> [(Int,Polynom a)]
appSff = appFact sff
```

GalFld/Algorithmen/Berlekamp.hs

```
appBerlekamp :: (Show a, FiniteField a, Num a, Fractional a) =>
                                        [(Int,Polynom a)] -> [(Int,Polynom a)]
appBerlekamp = appFact berlekampFactor
```

Wie bereits erwähnt wird auf die konkreten Algorithmen erst in Kapitel 3 eingegangen.

Zusammenfassen von Faktorisierungen Es ist sicherlich leicht vorstellbar, dass durch Anwendung von app$A$ verschiedene Tupel entstehen, deren eigentlicher Faktor jedoch der gleiche ist. `aggFact` ermöglicht die Zusammenfassung dieser Tupel nach Anwendung der Faktorisierungalgorithmen.

GalFld/Core/Factorization.hs

```
-- |Fasst in einer Faktorisierung gleiche Faktoren zusammen
aggFact :: (Num a, Eq a) => [(Int,Polynom a)] -> [(Int,Polynom a)]
aggFact l = [(sum [i | (i,g) <- l , f==g],f)
              | f <- nub [f | (_,f) <- l]
              , f /= pKonst 1]
```

2.6 Weiteres

2.6.1 Die Klasse ShowTex

Im Modul `GalFld.Core.ShowTex` wird eine Klasse `ShowTex` implementiert, welche es entsprechenden Datentypen mit dieser Klasse ermöglicht, nach LaTeX zu rendern.

Die einzige zur Klasse gehörende Funktion ist `showTex`, welche einen String mit LaTeX-Code zurückgibt.

Ferner wurden Funktionen implementiert, die

- einen Datentyp der Klasse `ShowTex` bzw.
- einen String mit LaTeX-Code in ein `PNG`-Bild umwandeln,
- sowie eine Funktion, die das Programm `sxiv` nutzt, um die erzeugten `PNG`-Bilder anzuzeigen.

Diese drei Funktionen sind nur unter Linux verfügbar.

GalFld/Core/ShowTex.hs

```haskell
-- |Wie renderRawTex, nur dass zunächst ShowTex aufgerufen wird.
renderTex :: (ShowTex a) => a -> IO ()
renderTex = renderRawTex . showTex
--------------------------------------------------------------------------------
-- |Nutze sxiv um das erzeugte Bild anzuzeigen
viewRendered = do createProcess (shell ("sxiv " ++ outputPNG))
                  return ()
#else
outputPNG = undefined
renderTex = undefined
renderRawTex = undefined
viewRendered = undefined
#endif
```

GalFld/Core/ShowTex.hs

```haskell
-- |Nimmt einen Latex-String und packt diesen in ein minimales Latex-Dokument,
-- rendert dieses und wandelt es danach in ein Bild um, wobei unnötiger Rand
-- entfernt wird.
renderRawTex :: String -> IO ()
renderRawTex x = do createProcess (shell cmd)
                    return ()
  where cmd = "latex -halt-on-error -output-directory " ++ outputDIR ++ " "
              ++ "'\\documentclass[12pt]{article}"
              ++ "\\pagestyle{empty}" ++ "\\usepackage{amsmath}"
              ++ "\\begin{document}"
              ++ "\\begin{multline*}" ++ x ++ "\\end{multline*}"
              ++ "\\end{document}' > /dev/null ; "
              ++ "dvipng -gamma 2 -z 9 -T tight -bg White " -- -bg Transparent
              ++ "-o " ++ outputPNG ++ " " ++ outputDVI ++ " > /dev/null"
--------------------------------------------------------------------------------
```

```
-- |Nutze sxiv um das erzeugte Bild anzuzeigen
viewRendered = do createProcess (shell ("sxiv " ⧺ outputPNG))
                  return ()
#else
outputPNG = undefined
renderTex = undefined
renderRawTex = undefined
viewRendered = undefined
#endif
```

2.6.2 Serialisierung

Alle hier genutzten Datentypen haben eine Instanz vom Typ `Binary`, welche einfache Serialisierung ermöglicht. Dazu müssen nur die zwei Funktionen `put` und `get` implementiert werden.

Als Beispiel dient folgende Funktion, der ein Dateiname als `String` und eine Liste von Polynomen über einer Erweiterung von einer Erweiterung von `PF` übergeben wird.

```
import qualified Data.Binary as B

saveToFile ∷ String → [Polynom (FFElem (FFElem PF))] → IO ()
saveToFile fileName polys = writeFile fileName (B.encode (polys))
```

Die erzeugte Binärdatei lässt sich ebenso einfach wieder auslesen, indem man die Daten in eine Variable `raw` lädt und diese mittels des Befehls `decode` decodiert. Dabei muss der Typ der einzulesenden Rohdaten spezifiziert werden.

```
readFromFile ∷ String → IO [Polynom (FFElem (FFElem PF))]
readFromFile fileName = do
  raw ← readFile fileName
  let ls = B.decode raw ∷ [Polynom (FFElem (FFElem PF))]
  return ls
```

2.6.3 Spezielle Polynome und zahlentheoretische Funktionen

`GalFld/More/SpecialPolys.hs` `GalFld/More/NumberTheory.hs`

Vorgreifend auf Kapitel 4 wird hier der Inhalt von `SpecialPolys.hs` beschrieben. In dieser Datei wurden zwei spezielle Familien von Polynomen über endlichen Körpern implementiert: die Kreisteilungspolynome und die Pi-Polynome.

Die Kreisteilungspolynome

Die Kreisteilungspolynome lassen sich auf verschiedene Arten definieren. Hier zitieren wir [4, Abschnitt 4].

Definition 2.16 (Kreisteilungspolynom) Sei $\mathbb{F}_q$ ein endlicher Körper und $n \in \mathbb{N}$. Dann heißt für $d \mid q^n - 1$

$$\Phi_d(X) := \prod_{u \in C_d} (X - u)$$

d-tes Kreisteilungspolynom, wobei

$$C_d := \{v \in \mathbb{F}_{q^n}^* : \ \operatorname{ord}(v) = d\}$$

die Menge der d-ten primitiven Einheitswurzeln in $\mathbb{F}_q$ ist.[6]

Definition 2.17 (Möbius-Funktion) Seien $n \in \mathbb{N}^*$ und $n = \prod_{j=1}^{l} p_j^{a_j}$ seine Primfaktorzerlegung, so heißt

$$\mu(n) := \begin{cases} 1, & n = 1 \\ 0, & \exists j : a_j \geq 2 \\ (-1)^l, & a_j = 1 \ \forall j \end{cases}$$

Möbius-Funktion von n.

Proposition 2.18 *Sei $d \in \mathbb{N}^*$. Dann gilt:*

$$\Phi_d(X) = \prod_{n \mid d} (X^n - 1)^{\mu(\frac{d}{n})} .$$

Beweis. [4, Abschnitt 4]. □

Implementierung Damit ist klar, wie man die Kreisteilungspolynome effizient implementiert.

GalFld/More/NumberTheory.hs

```haskell
-- |Primfaktorzerlegung (enthält Vielfache!)
--  aus http://www.haskell.org/haskellwiki/99_questions/Solutions/35
primFactors :: Int → [Int]
primFactors 1 = []
primFactors n = let divisors = dropWhile ((≠ 0) . mod n)
                                [2 .. ceiling $ sqrt $ fromIntegral n]
                in let prime = if null divisors then n else head divisors
                   in (prime :) $ primFactors $ div n prime
```

[6] Es sei vorausgesetzt, dass der Leser/indem Leser die nicht explizit definierten Termini bekannt sind.

GalFld/More/NumberTheory.hs

```
isPrime ∷ Int → Bool
isPrime n = 1 ≡ length (primFactors n)
```

GalFld/More/NumberTheory.hs

```
-- |Teiler von n
divisors ∷ Int → [Int]
divisors n | n ≡ 1     = div'
           | otherwise = div' ⧺ [n]
  where div' = 1 : filter ((≡0) . rem n) [2 .. n `div` 2]
```

GalFld/More/NumberTheory.hs

```
-- |Möbius-Funktion μ mit
--   μ(n) = (-1)^k,falls n quadratfrei, k = #Primfaktoren, 0 sonst
moebFkt ∷ Int → Int
moebFkt n | facs ≡ nub facs && even (length facs) = 1
          | facs ≡ nub facs && odd (length facs)  = -1
          | otherwise                             = 0
  where facs = primFactors n
```

GalFld/More/SpecialPolys.hs

```
-- |Die Kreisteilungspolynome
--   gibt das n-te Kreisteilungspolynom über dem Körper dem e zu Grunde liegt
cyclotomicPoly ∷ (Show a, Fractional a, Num a, FiniteField a) ⇒
                                              Int → a → Polynom a
cyclotomicPoly 1 e  = pTupUnsave [(1,1),(0,-1)]
cyclotomicPoly n e
  | isPrime n = pTupUnsave $ map (λi → (i,1)) $ reverse [0..n-1]
  | otherwise = foldl (@/) numerator $ map fst $ filter (λ(_,m) → m≡(-1)) l
  where numerator = product $ map fst $ filter (λ(_,m)→m≡1) l
        l = [(pTupUnsave [(n `quot` d, 1), (0,-1)], moebFkt d) | d ← divisors n]
```

Die Pi-Polynome

Hachenberger zeigt in [6], wie man *alle* primitiven und normalen Elemente einer Körpererweiterung $\mathbb{F}_{q^n}$ über $\mathbb{F}_q$ als Nullstellen eines Polynoms finden kann.

Bemerkung 2.19 Wie man sich leicht überlegt, sind alle primitiven Elemente eines Körpers $\mathbb{F}_q$, also $u \in \mathbb{F}_q^*$ mit $\operatorname{ord} u = q-1$, gerade Nullstellen des $(q-1)$-ten Kreisteilungspolynoms. Auf ganz analoge Weise kann man die normalen Elemente, also diejenigen $u \in \mathbb{F}_{q^n}$, deren additive Ordnung $\operatorname{Ord}_q(u)$ gleich $X^n - 1$ ist, als Nullstellen der Pi-Polynome schreiben.

Definition 2.20 (q-Polynom, [6, Definition 1.2]) Sei $f(X) = \sum_{i=0}^n f_i X^i \in \mathbb{F}_q[X]$, so heißt

$$F(X) := \sum_{i=0}^{n} f_i X^{q^i}$$

das *zu f assoziierte q-Polynom*.

Definition 2.21 (Pi-Polynom, [6, Definition 3.4]) Sei $f \in \mathbb{F}_q[X]$ ein monischer Teiler von $X^n - 1$. Notiere

$$A_f := \{u \in \mathbb{F}_{q^n} : \ \mathrm{Ord}(u) = f\}.$$

Dann heißt

$$P_f(X) := \prod_{v \in A_f} (X - v)$$

das *Pi-Polynom zu f über* $\mathbb{F}_q$.

Definition 2.22 Für $f, g \in \mathbb{F}_q[X]$ definiere

$$(f \odot g)(X) := f(G(X)).$$

Eine rekursiver Algorithmus zur Berechnung der Pi-Polynome ist dann nach [6, Abschnitt 4] gegeben durch:

Algorithmus 2.6: Berechnung Pi-Polynom

Input: $f(X) \in \mathbb{F}_q[X]$.
Output: $P_f(X) \in \mathbb{F}_q[X]$.
Algorithmus PIPOLY(f):
1. Berechne die vollständige Faktorisierung von f:
   $f(X) = \prod_{i=1}^{k} f_i^{\nu_i}$.
2. Setze $P_{f_1} := F_1(X)X^{-1}$.
3. Berechne $P_{f_1 \dots f_k}$ rekursiv durch
   $P_{f_1 \dots f_i} := (P_{f_1 \dots f_{i-1}} \odot f_i) P^{-1}_{f_1 \dots f_{i-1}}$
4. Setze $P_f := P_{f_1 \dots f_k} \odot (\prod_{i=1}^{k} f_i^{\nu_i - 1})$

Implementierung Man kann offenbar Algorithmus 2.6 direkt in Haskell übertragen.

GalFld/More/SpecialPolys.hs

```haskell
-- | Gibt das Pi-Polynom zu f
--   f muss ein monischer Teiler von x^m -1 über F_q sein
piPoly :: (Show a, Num a, Fractional a, FiniteField a) =>
                                            Polynom a -> Polynom a
piPoly f
  | isSqfree  = piSqFree
  | otherwise = piSqFree `odot` fNonSqFree
  where -- P_(tau f), wobei tau f der quadratfreie Teil von f ist
        piSqFree = foldl (\p f -> (p `odot` f) @/ p) pFst (map snd $ tail facs)
        -- Faktorisierung von f
        facs = factorP f
        -- Start der Rekursion mit P_(f1)
```

```
        pFst = assozPoly (snd $ head facs) @/ pTupUnsave [(1,1)]
        -- Definition von odot
        odot f g = evalPInP f $ assozPoly g
        -- Test auf quadratfrei
        isSqfree = all (≡1) $ map fst facs
        -- f / tau(f)
        fNonSqFree = unFact $ map (A.first (λi → i-1)) facs
```

3 Algorithmen auf Polynomen über endlichen Körpern

Über endlichen Körpern existieren verschiedene Ansätze, um ein Polynom zu faktorisieren. Diese sollen nun im Folgenden erläutert werden.

3.1 Quadratfreie Faktorisierung

Wir beginnen mit der Beschreibung eines Algorithmus zur quadratfreien Faktorisierung. Dazu sei im Folgenden k ein beliebiger Körper. Als Referenz dieses Abschnitts sei [2, Section 9] und [3, Section 8.3] genannt.

Definition 3.1 (quadratfrei, quadratfreier Teil) Seien $f(X) \in k[X]$ und seine vollständige Faktorisierung durch

$$f(X) = \prod_{i=1}^{d} f_i(X)^{\nu_i}$$

gegeben. Der *quadratfreie Teil von* $f(X)$ ist

$$\nu(f(X)) = \prod_{i=1}^{d} f_i(X)\,.$$

Ferner heißt $f(X)$ *quadratfrei*, falls

$$\nu(f(X)) = f(X)\,.$$

Definition 3.2 (quadratfreie Faktorisierung) Sei $f(X) \in k[X]$. Dann heißt

$$f(X) = c\prod_{i=1}^{m} r_i(X)^i$$

quadratfreie Faktorisierung von $f(X)$, falls für alle $i = 1, \ldots, m$ gilt, dass $r_i(X)$ monisch und quadratfrei ist und für alle $i, j = 1, \ldots, m$, $i \neq j$, stets $r_i(X)$ und $r_j(X)$ paarweise teilerfremd sind.

Bekanntlich ist für jedes nichttriviale Polynom $f(X)$ über einem Körper der Charakteristik Null $\mathrm{ggT}(f(X), f'(X)) \neq 0$, wobei $f'(X)$ die formale Ableitung von f meint. Damit kann man folgern, dass $\mathrm{ggT}(f(X), f'(X)) = 1$ genau dann, wenn $f(X)$ quadratfrei ist. (vgl. [2, Theorem 9.4, 2, Theorem 9.5]) Über endlichen Körpern geht dies nicht so einfach, wie folgendes Beispiel zeigt:

Beispiel 3.3 Sei $f(X) = X^3 + 1 \in \mathbb{F}_3[X]$. Dann ist

$$f'(X) = 3X^2 = 0\,.$$

Dennoch besitzt $f(X)$ eine quadratfreie Faktorisierung, da

$$f(X) = (x+1)^3\,.$$

3.1.1 Algorithmus zur quadratfreien Faktorisierung über endlichen Körpern

Einen Algorithmus zur quadratfreien Faktorisierung über Körpern der Charakteristik 0 findet man beispielsweise in [2, Figure 9.1] oder [3, Algorithm 8.1].

Für den passenden Algorithmus über endlichen Körpern halten wir uns an [3, Section 8.3]. Dazu starten wir mit der wesentlichen Aussage, die gerade in dem Fall, dass die Ableitung eines Polynoms 0 ist, die entscheidende Information liefert. Doch dies gilt nicht für beliebige Körper!

Definition 3.4 (perfekter Körper) Ein Körper $\mathbb{F}$ heißt *perfekt*, falls $\operatorname{char}\mathbb{F} = p$ für eine Primzahl p und der Frobenius $\sigma : \mathbb{F} \to \mathbb{F},\ x \mapsto x^p$ ein Automorphismus ist.

Proposition 3.5 *Seien $\mathbb{F}$ ein perfekter Körper und $f(X) \in \mathbb{F}[X]$. Ist $f'(X) = 0$, so existiert ein $b(X) \in \mathbb{F}[X]$ mit*

$$f(X) = (b(X))^p\,.$$

Beweis. Sei f gegeben als $f(X) = a_nX^n + \ldots + a_0$, so gilt offensichtlich durch Betrachtung der Definition und Regeln der formalen Ableitung, dass jede auftauchende Potenz von X ein Vielfaches von p sein muss. Also ist

$$f(X) = a_{pk}X^{pk} + \ldots + a_pX^p + b_0\,.$$

Definiere nun

$$b(X) = b_kX^k + \ldots + b_1X + b_0 \quad \text{mit} \quad b_i = a_{pi}^{\frac{1}{p}}\ i = 0, \ldots, k\,.$$

Da wir wissen, dass der Frobenius $\mathbb{F} \to \mathbb{F}, x \mapsto x^p$ ein Automorphismus auf $\mathbb{F}$ ist, ist $(\_)^{\frac{1}{p}}$ ein wohldefinierter Ausdruck und es gilt

$$f(X) = b(X)^p\,.$$

□

Beispiel 3.6 Sei $F = \mathbb{F}_p(z)$ für $z \notin \mathbb{F}_p$. Dann gilt für $f(X) = X^p - z \in F[X]$, dass $f'(X) = 0$, aber $f(X)$ ist über F irreduzibel. Offenbar besitzt z kein Urbild unter dem Frobenius; mithin ist F nicht perfekt.

Damit können wir nun einen Algorithmus zur quadratfreien Faktorisierung über endlichen Körpern formulieren.

Algorithmus 3.1: Quadratfreie Faktorisierung über endlichen Körpern

Input: $f(X) \in \mathbb{F}_q[X]$ monisch, $q = p^n$ eine Primzahlpotenz.
Output: $f(X) = r(X) = \prod_{i=1}^m r_i(X)^i$ quadratfreie Faktorisierung.
Algorithmus SFF($f(X)$):
$i := 1$, $r(X) := 1$, $b(X) := f'(X)$.
if $b(X) \neq 0$ **then** (1)
  $c(X) := \mathrm{ggT}(f(X), b(X))$
  $w(X) := f(X)/c(X)$
  while $w(X) \neq 1$ **do** (1.1)
    $y(X) := \mathrm{ggT}(w(X), c(X))$, $z(X) := w(X)/y(X)$
    $r(X) := r(X) \cdot z(X)^i$, $i := i + 1$
    $w(X) := y(X)$, $c(X) := c(X)/y(X)$
  endwhile
  if $c(X) \neq 1$ **then** (1.2)
    $c(X) := c(X)^{\frac{1}{p}}$
    $r(X) := r(X) \cdot (\mathrm{SFF}(c(X)))^p$
  endif
else (2)
  $f(X) := f(X)^{\frac{1}{p}}$
  $r(X) := (\mathrm{SFF}(f(X)))^p$
endif

Satz 3.7 *Algorithmus 3.1 berechnet die quadratfreie Faktorisierung für Polynome über endlichen Körpern (sogar über perfekten Körpern).*

Beweis. Sei $f(X)$ gegeben durch seine vollständige Faktorisierung in irreduzible Faktoren

$$f(X) = \prod_{i=1}^{d} f_i(X)^{\nu_i}$$

Schritt (2). Beginnen wir mit dem kürzeren Fall. Ist $b(X) = f'(X) = 0$, so existiert nach Proposition 3.5 eine p-te Wurzel des Polynoms. Auf diese lässt sich dann der Algorithmus rekursiv anwenden.

Schritt (1). Kommen wir nun zu dem Fall, wo auch wirklich etwas zu tun ist. Sei zunächst eine quadratfreie Faktorisierung von $f(X)$ gegeben, d.h.

$$f(X) = \prod_{i=1}^{m} a_i(X)^i$$

mit a_i quadratfrei und paarweise teilerfremd. Nun ist

$$b(X) = f'(X) = \sum_{i=1}^{m} a_1(X) \cdot \ldots \cdot a_{i-1}(X)^{i-1} \cdot i a_i(X)^{i-1} a_i'(X) \cdot a_{i+1}(X)^{i+1} \cdot \ldots \cdot a_m(X)^m$$

und wir können folgern,

$$c(X) = \mathrm{ggT}(f(X), f'(X)) = \prod_{i \in A} a_i(X)^{i-1}\,,$$

wobei $A = \{i = 1, \ldots, m : \ p \nmid i\}$. Es ist klar, dass diejenigen a_i, deren Exponent i ein Vielfaches der Charakteristik ist, nicht mehr im ggT auftauchen. Damit haben wir

$$w(X) = \frac{f(X)}{c(X)} = \prod_{i \in A} a_i(X)$$

ein Produkt der quadratfreien Faktoren in A mit jeweils einfacher Vielfachheit. Dieses können wir nun nutzen, um diese quadratfreien Faktoren zu isolieren: Für $A = \{i_1, \ldots, i_k\}$ in aufsteigend sortierter Reihenfolge haben wir

$$y(X) = \mathrm{ggT}(w(X), c(X)) = \prod_{j=i_1}^{i_k} a_j(X)\,,$$

$$z(X) = \frac{w(X)}{y(X)} = a_{i_1}(X)\,.$$

Nun ist klar, dass man die weiteren Faktoren, deren Exponenten in A liegen, durch iterative Anwendung dieser Idee erhält, wie man im Algorithmus erkennen kann. Letztlich bleibt nur die Frage, wie man an die Faktoren kommt, deren Exponenten Vielfache der Charakteristik sind. Dies ist aber offensichtlich, betrachtet man erneut Proposition 3.5 und die Umsetzung in Schritt (1.2). □

3.2 Der Algorithmus von Berlekamp

Sei im Folgenden $\mathbb{F}_q$ ein endlicher Körper von Charakteristik p und $f(X) \in \mathbb{F}_q[X]$ monisch. Ziel dieses Abschnittes ist ein Algorithmus, der eine vollständige Faktorisierung von $f(X)$ über $\mathbb{F}_q$ angibt.

3.2.1 Idee

Die grundlegende Idee des Berlekamp-Algorithmus besteht in folgendem Lemma.

Lemma 3.8 *Es gilt*

$$X^q - X = \prod_{a \in \mathbb{F}_q} (X - a) \in \mathbb{F}_q[X] \,.$$

Beweis. [17, Theorem 6.1 mit Corollary 4.5]. □

Ersetzen wir X durch ein Polynom $g(X) \in \mathbb{F}_q[X]$, so erhalten wir

$$g(X)^q - g(X) = \prod_{a \in \mathbb{F}_q} (g(X) - a)$$

und können uns nun überlegen, falls wir ein Polynom $g(X)$ mit $g(X)^q - g(X) \bmod f(X)$ haben, dann

$$f(X) \mid \prod_{a \in \mathbb{F}_q} (g(X) - a) \,,$$

was zumindest eine teilweise Faktorisierung von $f(X)$ liefern könnte.

Dass dies in der Tat funktioniert, zeigt nachstehender Satz, der obige Idee zusammenfasst und konkretisiert.

Satz 3.9 *Sei $g(X) \in \mathbb{F}_q[X]$ mit*

$$g(X)^q \equiv g(X) \bmod f(X) \,,$$

so gilt

$$f(X) = \prod_{a \in \mathbb{F}_q} \mathrm{ggT}(f(X), g(X) - a)$$

und die ggT*s sind paarweise teilerfremd.*

Beweis. [17, Theorem 9.1]. □

3.2.2 Die Berlekamp-Algebra

Damit haben wir nun eine Motivation für folgende Definition.

Definition 3.10 Der *Berlekamp-Raum zu* $f(X)$ bzw. die *Berlekamp-Algebra zu* $f(X)$ ist

$$\mathcal{B}_f := \{h(X) \in \mathbb{F}_q[X] : \deg h < \deg f, h(X)^q \equiv h(X) \bmod f(X)\}.$$

Bemerkung 3.11 In der Tat wird $\mathcal{B}_f$ zu einer $\mathbb{F}_q[X]/(f(X))$-Algebra.

Nun können wir ein wesentliches Resultat zitieren:

Satz 3.12 *Es gilt:*

$$\dim_F(\mathcal{B}_f) = s,$$

wobei s *die Anzahl irreduzibler paarweise verschiedener Faktoren von* $f(X)$ *ist.*

Beweis. [5, Satz 6.2]. □

Nun stellt sich natürlich die Frage, wie man konkret Elemente aus der Berlekamp-Algebra zu $f(X)$ findet. Blickt man noch einmal auf die Definition von $\mathcal{B}_f$, so erkennt man, dass $\mathcal{B}_f$ gerade der Kern folgender linearen Abbildung ist:

$$\begin{array}{rrcl} \Gamma_f : & \mathbb{F}_q[X]_{<\deg f} & \to & \mathbb{F}_q[X]_{<\deg f}, \\ & g(X) & \mapsto & g(X)^q - g(X) \bmod f(X). \end{array}$$

Nun können wir aber eine Darstellungsmatrix von Γ_f angeben, da wir die kanonische Basis von $\mathbb{F}_q[X]_{<\deg f}$ angeben können durch

$$\{1, X, X^2, \ldots, X^{\deg f - 1}\}.$$

3.2.3 Der Berlekamp-Algorithmus

Eine Kleinigkeit fehlt obigem Vorgehen noch, um daraus sicher eine teilweise Faktorisierung von $f(X)$ gewinnen zu können. Es ist a priori nicht klar, dass in Satz 3.9 die auftretenden ggTs eine echte, also nicht degenerierte, Faktorisierung von $f(X)$ liefern. Doch dies ist offensichtlich, falls $g(X) \in \mathcal{B}_f \setminus \mathbb{F}_q$ gewählt wird. Dann ist $\deg g < \deg f$ und daher $\mathrm{ggT}(f(X), g(X) - a) \neq f(X)$ für alle $a \in \mathbb{F}_q$.

Letztlich liefert noch nachstehendes Lemma die Grundlage für eine rekursive Anwendung des Algorithmus:

Lemma 3.13 *Ist $a(X) \in \mathbb{F}_q[X]$ monisch mit $a(X) \mid f(X)$, so ist*

$$\begin{array}{rrcl} \pi: & \mathcal{B}_f & \to & \mathcal{B}_a \\ & g(X) & \mapsto & g(X) \bmod a(X) \end{array}$$

eine surjektive lineare Abbildung.

Beweis. klar. □

Implementierung

Um tatsächlich eine *vollständige* Faktorisierung zu erhalten, muss man sich noch überlegen, dass dies mit Hilfe des Berlekamp-Algorithmus nur möglich ist, falls $f(X)$ quadratfrei ist (vgl. Satz 3.12!). Daher sei im Folgenden $f(X)$ stets ein quadratfreies, monisches Polynom über $\mathbb{F}_q[X]$.

Berechnung einer Basis von $\boldsymbol{\mathcal{B}_f}$ Die Basis des Berlekampraumes berechnen wir mit den in Abschnitt 2.4 vorgestellten Methoden der linearen Algebra.

GalFld/Algorithmen/Berlekamp.hs

```haskell
-- |Berechnet eine Basis des Berlekampraums zu f,
--  d.h. gibt eine Matrix zurück, deren Zeilen gerade den Berlekampraum
--  aufspannen bzgl der kanonischen Basis { 1, x, x^2, x^3, ... }
berlekampBasis :: (Show a, Fractional a, Num a, FiniteField a)
                                                      => Polynom a -> Matrix a
berlekampBasis f = transposeM $ kernelM $ transposeM $!
                        fromListsM [red i | i <- [0..(n-1)]] - genDiagM 1 n
  where !n     = fromJust $ degP f
        !q     = elemCount a
        !a     = getReprP f
        {-# INLINE red #-}
        red i = takeFill 0 n $ p2List $ modByP (pTupUnsave [(i*q,1)]) f
```

Die Funktion `red i` liefert dabei gerade das `i`-te Basiselement der kanonischen Basis von $\mathbb{F}_q[X]/(f(X))$.

Der Berlekamp-Algorithmus Bei einer konkreten Umsetzung des Berlekamp-Algorithmus bleibt immer die Frage, wie ein Element $g(X) \in \mathcal{B}_f \setminus \mathbb{F}_q$ zu wählen ist. Wir haben uns entschieden, stets das zweite Basiselement (das erste ist immer 1) zu wählen. Sicherlich könnte man auch ein zufälliges Element wählen; dies widerspricht aber des funktionalen Prinzips, das Haskell zugrunde liegt.

GalFld/Algorithmen/Berlekamp.hs

```haskell
-- |Faktorisiert ein Polynom f über einem endlichen Körper
--  Voraussetzungen: f ist quadratfrei
--  Ausgabe: Liste von irreduziblen, pw teilerfremden Polynomen
berlekampFactor ∷ (Show a, Fractional a, Num a, FiniteField a)
                                                   ⇒ Polynom a → [(Int,Polynom a)]
berlekampFactor f | isNullP f   = []
                  | uDegP f < 2 = [(1,f)]
                  | otherwise   = berlekampFactor' f m
  where !m = berlekampBasis f
        {-# INLINE berlekampFactor' #-}
        berlekampFactor' ∷ (Show a, Num a, Fractional a, FiniteField a)
                                            ⇒ Polynom a → Matrix a → [(Int,Polynom a)]
        berlekampFactor' f m | uDegP f ≤ 1       = [(1,f)]
                             | getNumRowsM m ≡ 1 = [(1,f)]
                             | otherwise         =
                              berlekampFactor' g n ⧺ berlekampFactor' g' n'
          where {-# INLINE g #-}
                g  = head [x | x ← [ggTP f (h - pKonst s)
                                | s ← elems (getReprP f)] , x ≠ 1]
                {-# INLINE g' #-}
                g' = f @/ g
                {-# INLINE h #-}
                h  = pList $ getRowM m 2
                {-# INLINE n #-}
                n  = newKer m g
                {-# INLINE n' #-}
                n' = newKer m g'
                {-# INLINE newKer #-}
                newKer m g  = fromListsM $! take r m'
                  where !(k,l) = boundsM m
                        !m'    = toListsM $ echelonM $ fromListsM
                                  [takeFill 0 l $ p2List $
                                   modByP (pList (getRowM m i)) g | i ← [1..k]]
                        !r     = k-1- fromMaybe (-1) (findIndex (all (≡0))
                                                        $ reverse m')
```

Die Berechnung der neuen Basis bei der rekursiven Anwendung ist aufgrund Lemma 3.13 relativ einfach, da π simplerweise auf die schon vorhandene Berlekampbasis angewendet werden kann.

Beispiel 3.14 Angenommen wir wollen das Polynom

$$X^5 + X^4 + 3X^3 + 3X^2 + 2X + 2 \ \in \mathbb{F}_5[X]$$

faktorisieren. Wir berechnen also

$$\begin{aligned}
1^5 &\equiv 1 && \bmod f(X)\\
X^5 &\equiv 4X^4+2X^3+2X^2+3X+3 && \bmod f(X)\\
X^{10} &\equiv X^2 && \bmod f(X)\\
X^{15} &\equiv 2X^4+2X^3+X^2+X+4 && \bmod f(X)\\
X^{20} &\equiv X^4 && \bmod f(X)
\end{aligned}$$

und erhalten damit eine Darstellungsmatrix von Γ bezüglich der Basis $\{1, X, X^2, X^3, X^4\}$ von $\mathbb{F}_5[X]_{<5}$ und können diese in Zeilenstufenform bringen:

$$D_\Gamma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0\\ 3 & 3 & 2 & 2 & 4\\ 0 & 0 & 1 & 0 & 0\\ 4 & 1 & 1 & 2 & 2\\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{bmatrix} 1 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 3 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Also ist eine Basis von $\mathcal{B}_f$ gegeben durch

$$B_f := \{1, X^3+X, X^2, X^4\}.$$

Wir wählen – wie oben beschrieben – das zweite Basiselement $h(X) = X^3 + X$ aus und berechnen

$a \in \mathbb{F}_q$	0	1	2	3	4
$\mathrm{ggT}(f(X), h(X)-a)$	X^2+2	X^2+4X+3	1	1	$X+2$

Dies erlaubt nun iterative Anwendung des Berlekamp-Algorithmus, nämlich für X^2+2, X^2+4X+3 und für $X+2$. Letzteres ist natürlich offensichtlich irreduzibel.

Für $f_1(X) := X^2+2$ haben wir

$$B_f \bmod f_1(X) = \{1, 0, 3, 4\}$$

und für $f_2(X) := X^2+4X+3$

$$B_f \bmod f_2(X) = \{1, 1, X+2, 1\}.$$

Für f_1 bricht der Berlekamp-Algorithmus sofort ab, da offenbar die Dimension des Berlekampraumes 1 ist. Für f_2 wählen wir $h(X) = X+2$ und erhalten

$a \in \mathbb{F}_q$	0	1	2	3	4
$\mathrm{ggT}(f_2(X), h(X)-a)$	1	1	$X+3$	1	$X+1$

Damit ist die vollständige Faktorisierung von $f(X)$ über $\mathbb{F}_5$ bekannt:

$$f(X) = (X+1)(X+2)(X+3)(X^2+2).$$

3.3 Irreduzibilitätstest nach Rabin

Der in Abschnitt 3.2 vorgestellte Algorithmus von Berlekamp faktorisiert (quadratfreie) Polynome stets vollständig. Jedoch kann man sich Anwendungen vorstellen, in denen lediglich interessant ist, ob ein Polynom irreduzibel ist oder nicht. Dabei würde eine Anwendung des Berlekamp-Algorithmus unnötige Arbeit leisten. Daher wollen wir das zentrale Resultat von Rabin aus [15] zitieren, das den gleichnamigen Algorithmus motiviert.

> **Satz 3.15** *Sei $f(X) \in \mathbb{F}_q[X]$ monisch von Grad n. Seien $p_1, \dots, p_k$ alle paarweise verschiedenen Primteiler von n. Notiere mit n_i den Kofaktor von p_i in n, also $n_i := \frac{n}{p_i}$ für $i = 1, \dots, k$. Dann gilt: $f(X)$ ist irreduzibel über $\mathbb{F}_q$ genau dann, wenn*
>
> 1. *$f(X) \mid (X^{q^n} - X)$,*
> 2. *$\mathrm{ggT}(g(X), X^{q^{n_i}} - X) = 1$ für alle $i = 1, \dots, k$.*

Beweis. [15, Lemma 1]. Dort zwar nur für $\mathbb{Z}_p$, jedoch lässt sich der Beweis für beliebiges $\mathbb{F}_q$ problemlos erweitern. □

Damit ist klar, wie man mit Satz 3.15 einen Irreduzibilitätstest gestaltet. Es gilt lediglich zu bemerken, dass die Berechnung des $\mathrm{ggT}(g(X), X^{q_i^n} - X)$ in dieser Form aufgrund des hohen Grades des zweiten Polynoms sehr schwierig wäre. Daher erfolgt zuvor eine Reduktion von $X^{q^{n_i}} - X \bmod f(X)$.

GalFld/Algorithmen/Rabin.hs

```
-- |Rabin's Irreduzibilitätstest
--  Ausgabe: True, falls f irreduzibel, False, falls f reduzibel
--
--
--  Algorithm Rabin Irreducibility Test
--  Input: A monic polynomial f in Fq[x] of degree n,
--            p1, ..., pk all distinct prime divisors of n.
--  Output: Either "f is irreducible" or "f is reducible".
--  Begin
--        for j = 1 to k do
--            n_j := n / p_j;
--        for i = 1 to k do
--            h := x^(q^n_i) - x  mod f;
--            g := gcd(f, h);
--            if g  1, then return 'f is reducible' and STOP;
--        end for;
--        g := x^(q^n) - x  mod f;
--        if g = 0, then return "f is irreducible",
--             else return "f is reducible"
--  end.
--  vgl http://en.wikipedia.org/wiki/Factorization_of_polynomials_over_
--        finite_fields#Irreducible_polynomials
rabin :: (Show a, FiniteField a, Num a, Fractional a, Eq a) ⇒ Polynom a → Bool
```

```haskell
rabin f | isNullP f = False
        | otherwise = rabin' f ns
  where ns = map (λp → d `quot` p) $ nub $ factor d
        d  = uDegP f
        q  = elemCount $ getReprP f
        pX = pTupUnsave [(1,1)]
        -- eigentlicher Rabin für den letzen Test mit x^(q^n) - x
        rabin' f [] = isNullP g
          where g  = (h'-pX) `modByP` f
                h' = modMonom q d f
        -- eigentlicher Rabin für x^(q^n_j) - x mit n_j = n / p_j
        rabin' f (n:ns) | g ≠ pKonst 1  = False
                        | otherwise = rabin' f ns
          where g  = ggTP f (h'-pX)
                h' = modMonom q n f
```

Zu beachten gilt, dass die Reduktion $\bmod f(X)$ nicht mit dem in Abschnitt 2.1 vorgestellten `modByP` durchgeführt wird, sondern eine separate Funktion implementiert wurde, die effizient durch einen *Divide-And-Conquer*-Ansatz $X^{q^{n_i}} \bmod f(X)$ berechnet.

GalFld/Algorithmen/Rabin.hs

```haskell
-- | Schnelles Modulo für Monome, d.h. berechnet
--    x^(q^d) mod f
modMonom :: (Show a, Num a, Eq a, Fractional a) ⇒
                                      Int → Int → Polynom a → Polynom a
modMonom q d  = modMonom' n
  where n  = toInteger q ^ toInteger d
        modMonom' n f
          | n < toInteger df
                      = pTupUnsave [(fromInteger n,1)]
          | even n    = g `modByP` f
          | otherwise = multMonomP 1 g `modByP` f
          where df = uDegP f
                m  = n `quot` 2
                g  = h*h
                h  = modMonom' m f
```

Die Zerlegung von n in seine Primfaktoren wird durch `nub . factor` bewerkstelligt, wobei `nub` aus `Data.List` Duplikate in Listen entfernt. `factor` ist gegeben durch:

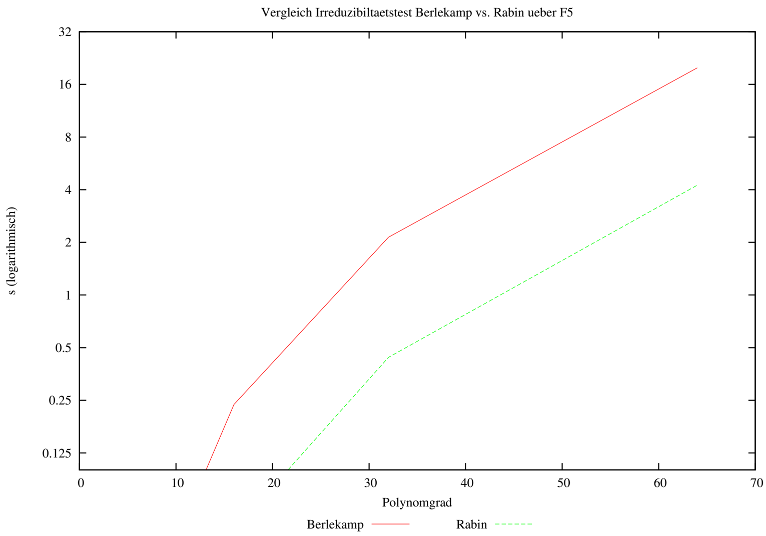
GalFld/Algorithmen/Rabin.hs

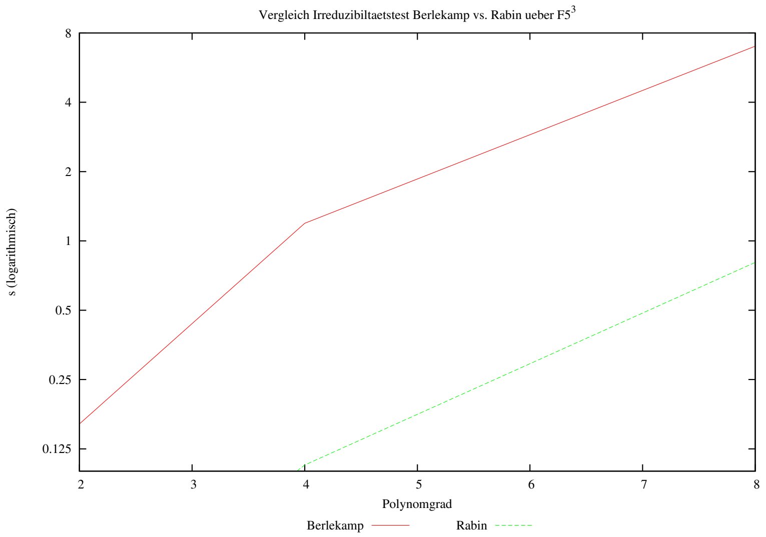
```haskell
-- |Primfaktorzerlegung
--   aus http://www.haskell.org/haskellwiki/99_questions/Solutions/35
factor :: Int → [Int]
factor 1 = []
factor n = let divisors = dropWhile ((≠ 0) . mod n) [2 .. ceiling $ sqrt $ fromIntegral n]
           in let prime = if null divisors then n else head divisors
              in (prime :) $ factor $ div n prime
```

3.3.1 Ein kleiner Vergleich

Auch wenn der Vergleich einer vollständigen Faktorisierung via Berlekamp mit Rabin als Irreduzibilätstest ob des Mehraufwands nicht ganz fair ist, so wollen wir ihn doch anführen. Abbildung 3.1 zeigt deutlich, dass die bloße Information der Irreduzibilität viel leichter zu gewinnen ist, als die gesamte Faktorisierung. (Wer hätte das gedacht!)

Abbildung 3.1: Vergleich Berlekamp vs. Rabin als Irreduzibilitätstest





4 Beispiel: Suche nach primitiv-normalen Elementen

`examples/ExamplePrimitiveNormal.lhs`

Wir beginnen mit einer Körpererweiterung $\mathbb{F}_{q^n} \mid \mathbb{F}_q$ und stellen uns die Frage nach einer Enumeration aller primitiven und normalen Elemente dieser Erweiterung. Wie bereits in Bemerkung 2.19 erläutert, sind die Nullstellen des Pi-Polynoms zu $X^n - 1$ gerade die normalen Elemente der Körpererweiterung und die Nullstellen des Kreisteilungspolynoms Φ_{n-1} gerade die primitiven Elemente. Folglich ist der ggT beider gerade das Produkt der Minimalpolynome aller primitiven *und* normalen Elemente!

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
module Main
  where
```

Imports zum Messen der Ausführungszeit und zum Verarbeiten von Input-Parametern.

```haskell
import System.CPUTime
import System.Environment
```

Ferner benötigen wir die Bibliothek `GalFld` und `GalFld.More.SpecialPolys`.

```haskell
import GalFld.GalFld
import GalFld.More.SpecialPolys
```

Wir erzeugen einen Primkörper der Charakteristik 2 mit dem Namen `PF`.

```haskell
$(genPrimeField 2 "PF")

pf = 1::PF
p = charakteristik pf
```

Anschließend erstellen wir eine neue Datenstruktur, genannt `T`, die die gesammelten Informationen speichern soll.

```haskell
data T = T { ext :: Int -- Grad des Grundkörpers über dem Primkörper
           , deg :: Int -- Grad der Erweiterung
           , countP :: Int -- Anzahl primitiver Elemente
           , countN :: Int -- Anzahl normaler Elemente
           , countPN :: Int } -- Anzahl primitiv-normaler Elemente
```

Nach diesen Schritten der Vorbereitung können wir nun den zentralen Teil des Beispiels formulieren: Die Berechnung der primitiv-normalen Elemente durch Faktorisierung des ggT des Kreisteilungspolynoms und des passenden Pi-Polynoms.

```
genPrimNorm :: Int → Int → (T, [(Int, Polynom (FFElem PF))])
genPrimNorm m n = (record, fac)
  where one     = extendFFBy m pf
        cyP     = cyclotomicPoly (p^(n*m)-1) one
        piP     = piPoly $ pTupUnsave [(n,one),(0,-1)]
        ggT     = ggTP cyP piP
        fac     = factorP ggT
        record  = T m n (uDegP cyP) (uDegP piP) (uDegP ggT)
```

Bleibt nur noch `if'` als kleines Hilfsmittel zu formulieren

```
if' :: Bool → a → a → a
if' True  x _ = x
if' False _ y = y
```

und in einer `main`-Funktion die Ein- und Ausgaben zusammenzufügen.

```
main = do
  args ← getArgs
  let indxs = if' (length args ≡ 2)
                  [(read $ head args)..(read $ head $ tail args)]
                  ( if' (length args ≡ 1)
                        [2..(read $ head args)]
                        [2..] )
  mapM_ (λm → (mapM_ (λn → do
    st ← getCPUTime
    let gpn = genPrimNorm m n
    putInfo $ fst gpn
    --putPolys $ snd gpn
    putTime st ) indxs)) [2..5]
      where putInfo (T m n cP cN cPN) = do
              putStrLn $ "In F" ++ show (p^m) ++ "^" ++ show n ++ " über F" ++ show (p^m)
                ++ " gibt es:"
              putStrLn $ "\t\t" ++ show cP ++ " primitive Elemente"
              putStrLn $ "\t\t" ++ show cN ++ " normale Elemente"
              putStrLn $ "\t\t" ++ show cPN ++ " primitive und normale Elemente"
            putPolys fs = do
              putStrLn "Mit Minimalpolynomen:"
              mapM_ (λ(_,f) → putStrLn $ "\t" ++ show f) fs
            putTime st = do
              ft ← getCPUTime
              putStrLn $ "("
                ++ show (fromIntegral (ft - st) / 1000000000000) ++ "s)\n"
```

Die berechneten Werte sind in Anhang A aufgelistet.

5 Ausblicke

In den vorherigen Kapiteln konnten wir die Umsetzung einer Bibliothek von Grundfunktionen auf endlichen Körpern in Haskell erläutern und demonstrieren. Sicherlich sind die bisher implementierten Funktionen bei weitem nicht ausreichend, um dieses Library als vollständig bezeichnen zu können. So ist klar, dass die meisten Computer-Algebra-Systeme, was den Funktionsumfang endlicher Körper betrifft, unserem kleinen Softwareprojekt überlegen sind. Es gilt jedoch zu bemerken, dass wir gerade auf den Funktionsumfang für Polynome über endlichen Körpern, insbesondere was verschiedene Multiplikations- und Faktorisierungsalgorithmen angeht, besonderes Augenmerk gelegt haben.

Wie könnte es weitergehen?

Statt einer Schlussbemerkung drängt sich daher sicherlich die Frage nach einer Fortsetzung des Projekts auf.

Erweiterung des Funktionsumfangs Das Hinzufügen neuer Funktionen könnte das Projekt fortsetzen. Bekanntlich existieren gerade für endliche Körper der Charakteristik 2 spezielle Algorithmen, die weitaus effizienter sind, als ihre Pendents in allgemeiner Charakteristik. Aus hauptsächlich mathematisch interessierter Sicht ist dies vermutlich eine spannende Aufgabe, da – wie wir im Laufe des Projekts erkennen konnten – die Syntax von Haskell der Art und Weise mathematischer Notation besonders ähnlich ist.

Performance der Implementierung Trotz der „Schönheit“ funktionaler Programmierung mussten wir an vielen Stellen bemerken, dass das Konzept der unveränderlichen Objekte auf Kosten der Performance geht. Insbesondere bei großen Datenstrukturen, wie z.B. Polynomen über Körpererweiterungen (also Polynome, deren Koeffizienten wiederum Polynome sind) oder auch Matrizen mit polynomialen Einträgen, nimmt der *garbage collector*, also dasjenige Unterprogramm der Ausführung, das den Speicher nicht mehr benutzter Objekte wieder frei gibt, einen großen, wenn nicht sogar den größten Teil der Ausführungszeit ein. An diesem Punkt besteht sicherlich großes Optimierungspotential. Ein Ansatz könnte es sein, an den berechnungsintensiven Stellen von der Funktionalität abzuweichen und in *Monaden*[1] (z.B. [7]) zu wechseln. Alternativ könnte man natürlich versuchen, *mehr* statt weniger Funktionalität zur Verbesserung der Laufzeit einzusetzen. Da, wie bereits öfters erwähnt, Haskell Ausdrücke nicht auswertet, solange sie nicht de facto benötigt werden, könnte man versuchen, die entstehenden *Thunk*s (z.B. [9]), also die noch nicht ausgewerteten Stellen, zu vereinfachen. Beide Herangehensweisen sind sicherlich legitim, würden jedoch eine weitaus intensivere Einarbeitung in die tiefe Struktur von Haskell erfordern und den Rahmen dieses Projekts sprengen.

Obwohl an vielen Stellen sicherlich noch Optimierungspotential bezüglich der Performance des Projekts besteht, möchten wir anmerken, dass Haskell im Allgemeinen mindestens genauso schnell ist, wie andere Programmiersprachen. (vgl. [8])

Nun möchten wir das Wort an Philip Greenspun und Autrijus Tang übergeben, um abschließend ein Gefühl für das Programmieren in Haskell zu geben.

> *SQL, Lisp, and Haskell are the only programming languages that I've seen where one spends more time thinking than typing.* Philip Greenspun[2]

> *Haskell is faster than C++, more concise than Perl, more regular than Python, more flexible than Ruby, more typeful than C#, more robust than Java, and has absolutely nothing in common with PHP.* Audrey Tang[3]

[1] Übrigens sind Monaden aus Sicht der Kategorientheorie sehr interessante Objekte.
[2] `http://blogs.law.harvard.edu/philg/2005/03/07/how-long-is-the-average-internet-discussion-forum-posting/`
[3] `http://www.perl.com/pub/2005/09/08/autrijus-tang.html?page=2`

A Berechnete Werte

A.1 Erweiterungen über Primkörpern

Die folgenden Tabellen sind nach Charakteristik der Körper sortiert. Diese Charakteristik wird mit p bezeichnet. In jeder Tabelle gibt es zu verschiedenen Körpererweiterungen, welche mit n bezeichnet werden, je eine Zeile. In jeder Zeile stehen nacheinander die Anzahl der primitiven Elemente, die Anzahl der normalen und die Anzahl derer, die beide Eigenschaften besitzen.

Im Kapitel A.3 ist der dazu verwendete Quellcode gelistet.

Tabelle A.1: Werte für $p = 2$

p	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
2	2	2	2	2
2	3	6	3	3
2	4	8	8	4
2	5	30	15	15
2	6	36	24	18
2	7	126	49	49
2	8	128	128	56
2	9	432	189	171
2	10	600	480	290
2	11	1936	1023	957
2	12	1728	1536	624
2	13	8190	4095	4095
2	14	10584	6272	4074
2	15	27000	10125	8430
2	16	32768	32768	16272
2	17	131070	65025	65025
2	18	139968	96768	51660
2	19	524286	262143	262143
2	20	480000	491520	225100
2	21	1778112	583443	495159
2	22	2640704	2095104	1319692

Tabelle A.2: Werte für $p = 3$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
3	2	4	4	4
3	3	12	18	9
3	4	32	32	16
3	5	110	160	75
3	6	288	324	144
3	7	1092	1456	728
3	8	2560	2048	832
3	9	9072	13122	6075
3	10	26400	25600	11520
3	11	84700	117128	55979
3	12	165888	209952	65424
3	13	797160	913952	456976
3	14	2384928	2119936	1057392

Tabelle A.3: Werte für $p = 5$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
5	2	8	16	8
5	3	60	96	48
5	4	192	256	64
5	5	1400	2500	1130
5	6	4320	9216	2568
5	7	39060	62496	31248
5	8	119808	147456	44928
5	9	894240	1499904	687132
5	10	2912000	6250000	1862760

Tabelle A.4: Werte für $p = 7$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
7	1	2	6	2
7	2	16	36	16
7	3	108	216	72
7	4	640	1728	480
7	5	5600	14400	4800
7	6	36288	46656	14832
7	7	264992	705894	227010
7	8	1536000	3981312	1062400

Tabelle A.5: Werte für $p = 11$

p	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
11	1	4	10	4
11	2	32	100	32
11	3	432	1200	384
11	4	3840	12000	3200
11	5	64400	100000	40000
11	6	373248	1440000	303456

Tabelle A.6: Werte für $p = 13$

p	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
13	1	4	12	4
13	2	48	144	48
13	3	720	1728	576
13	4	6144	20736	4352
13	5	123760	342720	114240
13	6	1347840	2985984	834048

Tabelle A.7: Werte für $p = 17$

p	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
17	1	8	16	8
17	2	96	256	96
17	3	2448	4608	2304
17	4	21504	65536	16896
17	5	709920	1336320	668160
17	6	6345216	21233664	5583168

Tabelle A.8: Werte für $p = 19$

p	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
19	1	6	18	6
19	2	96	324	96
19	3	2268	5832	1944
19	4	34560	116640	31104

Tabelle A.9: Werte für $p = 23$

p	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
23	1	10	22	10
23	2	160	484	160
23	3	4680	11616	4440
23	4	66560	255552	60640

Tabelle A.10: Werte für $p = 29$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
29	1	12	28	12
29	2	192	784	192
29	3	9504	23520	9180
29	4	161280	614656	139776

Tabelle A.11: Werte für $p = 31$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
31	1	8	30	8
31	2	256	900	256
31	3	7920	27000	7200
31	4	221184	864000	207360

Tabelle A.12: Werte für $p = 37$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
37	1	12	36	12
37	2	432	1296	432
37	3	14256	46656	13176
37	4	470016	1679616	420864

Tabelle A.13: Werte für $p = 41$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
41	1	16	40	16
41	2	384	1600	384
41	3	27552	67200	26880
41	4	623616	2560000	564224

Tabelle A.14: Werte für $p = 43$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
43	1	12	42	12
43	2	480	1764	480
43	3	22680	74088	21168

Tabelle A.15: Werte für $p = 47$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
47	1	22	46	22
47	2	704	2116	704
47	3	47520	101568	46596

Tabelle A.16: Werte für $p = 53$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
53	1	24	52	24
53	2	864	2704	864
53	3	58752	146016	57744

Tabelle A.17: Werte für $p = 61$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
61	1	16	60	16
61	2	960	3600	960
61	3	55296	216000	52848

Tabelle A.18: Werte für $p = 67$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
67	1	20	66	20
67	2	1280	4356	1280
67	3	75600	287496	72000

Tabelle A.19: Werte für $p = 71$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
71	1	24	70	24
71	2	1152	4900	1152
71	3	122688	352800	120960

Tabelle A.20: Werte für $p = 73$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
73	1	24	72	24
73	2	1728	5184	1728
73	3	129600	373248	124416

Tabelle A.21: Werte für $p = 79$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
79	1	24	78	24
79	2	1536	6084	1536
79	3	127008	474552	122040

Tabelle A.22: Werte für $p = 83$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
83	1	40	82	40
83	2	1920	6724	1920
83	3	263520	564816	260280

Tabelle A.23: Werte für $p = 89$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
89	1	40	88	40
89	2	1920	7744	1920
89	3	320400	696960	316800

Tabelle A.24: Werte für $p = 97$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
97	1	32	96	32
97	2	2688	9216	2688
97	3	304128	884736	294912

Tabelle A.25: Werte für $p = 101$

p	n	anz. primitiver	anz. normaler	anz. primitiv-normaler
101	1	40	100	40
101	2	2560	10000	2560
101	3	412080	1020000	408000

A.2 Erweiterungen über Nicht-Primkörpern

In diesem Abschnitt betrachten wir Erweiterungen von Nicht-Primkörpern, d.h. endliche Körper $\mathbb{F}_q$, wobei $q = p^m$ für eine Primzahl p und $m > 1$ gilt.

Tabelle A.26: Werte für $p = 2$ und $m = 2$

p	**m**	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
2	2	1	2	3	2
2	2	2	8	12	8
2	2	3	36	27	18
2	2	4	128	192	96
2	2	5	600	675	400
2	2	6	1728	1728	792
2	2	7	10584	11907	7784
2	2	8	32768	49152	24448
2	2	9	139968	107163	57186
2	2	10	480000	691200	316760

Tabelle A.27: Werte für $p = 2$ und $m = 3$

p	**m**	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
2	3	1	6	7	6
2	3	2	36	56	36
2	3	3	432	441	378
2	3	4	1728	3584	1512
2	3	5	27000	28665	23760
2	3	6	139968	225792	120744
2	3	7	1778112	823543	698544

Tabelle A.28: Werte für $p = 2$ und $m = 4$

p	**m**	**n**	**anz. primitiver**	**anz. normaler**	**anz. primitiv-normaler**
2	4	1	8	15	8
2	4	2	128	240	128
2	4	3	1728	3375	1440
2	4	4	32768	61440	30720
2	4	5	480000	759375	346400

A.3 Der Quellcode

A.3.1 Erweiterungen über Primkörpern

Hier das Bash-Script, das den Haskell Code generiert, der zur Berechnung der Werte genutzt wurde. Das Script compiliert und startet die erzeugten Programme danach parallel.

```
#!/bin/bash
################################################################################
# usage:
#     cPN.sh
#     cPN.sh TOP
#     cPN.sh TOP "PRIME1 PRIME2 ..."
#
# Last modified: Sun Sep 07, 2014  09:33

###############################################################################
# Config
PATHBASE="/tmp/cPN/"
if [ $# -eq 2 ]; then
  LIMIT=$1
  LIST=$2
elif [ $# -eq 2 ]; then
  LIST="2 3 5 7 11 13 17"
  LIMIT=$1
else
  LIST="2 3 5 7 11 13 17"
  LIMIT=8
fi

nCPUm1=3

###############################################################################
FAIL=0
mkdir -p $PATHBASE
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
mkdir -p "${DIR}/cPNout/"
SRC="${DIR}/../src/"

genPathHS(){
  echo "${SRC}cPN${1}.hs"
}
genPathEx(){
  echo "${PATHBASE}cPN${1}"
}
genPathCSV(){
  echo "${DIR}/cPNout/CalcPrimNubers_p=${1}.csv"
}

genHS(){
  # genHS CHARAKTERISTIK BEGINN END
```

```
cat <<HS
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE BangPatterns #-}
module Main
  where
import System.Environment
import System.Directory
import Control.Monad
import Data.Time

import GalFld.GalFld
import GalFld.More.SpecialPolys

p = ${1}
λ$(genPrimeField ${1} "PF")
outFile = "$(genPathCSV $1)"

pf = 1::PF

data T = T { deg :: Int -- Grad der Erweiterung
           , countP :: Int -- Anzahl primitiver Elemente
           , countN :: Int -- Anzahl normaler Elemente
           , countPN :: Int } -- Anzahl primitiv-normaler Elemente

genPrimNorm :: Int → (T, [(Int, Polynom PF)])
genPrimNorm n = (record, fac)
  where cyP    = cyclotomicPoly (p^n-1) pf
        piP    = piPoly $ pTupUnsave [(n,pf),(0,-1)]
        ggT    = ggTP cyP piP
        fac    = factorP ggT
        record = T n (uDegP cyP) (uDegP piP) (uDegP ggT)

main = do
  args ← getArgs
  let indxs = [(read $ head args)..(read $ head $ tail args)]
  bool ← doesFileExist outFile
  unless bool $ putHeaderToFile p
  mapM_ (λn → do
    let t = fst $ genPrimNorm n
    putInfo t p
    getCurrentTime >≥ print
    putToFile t p) indxs
      where putInfo !(T n cP cN cPN) p = do
              putStrLn $ "In F" ++ show p ++ "^" ++ show n ++ " über F" ++ show p
                ++ " gibt es:"
                ++ "\t" ++ show cP
                ++ "\t" ++ show cN
                ++ "\t" ++ show cPN
            putHeaderToFile p =
              writeFile outFile "p,n,#primitive,#normal,#primitivNormal\n"
            putToFile (T n cP cN cPN) p =
              appendFile outFile $ show p ++ ","
                                  ++ show n ++ ","
```

```
                                    # show cP # ","
                                    # show cN # ","
                                    # show cPN # "\n"

HS
}

getDoneNumber(){
  csv=$(genPathCSV $1)
  if [ -f $csv ]; then
    echo `expr $(tail -n1 $csv | cut -d',' -f2) + 1`
  else
    echo "1"
  fi
}

###############################################################################
# gen and compile HS
pushd $SRC
for p in $LIST; do
  if [ ! -f $(genPathEx $p) ]; then
    genHS $p >$(genPathHS $p)
    mkdir -p "${PATHBASE}compileOut/"
    ghc λ
      -outputdir "${PATHBASE}compileOut/" λ
      -o "$(genPathEx $p)" λ
      -O2 -isrc -threaded λ
      "$(genPathHS $p)"
    rm $(genPathHS $p)
  fi
done
popd

###############################################################################
for p in $LIST; do
  if [ -f $(genPathEx $p) ]; then
    #echo $(getDoneNumber $p)
    $(genPathEx $p) $(getDoneNumber $p) $LIMIT &
  fi

  # limit subprocesses to 4
  while [ $(jobs -p | wc -l) -gt $nCPUm1 ] ; do
    sleep 1
  done
done

###############################################################################
for job in `jobs -p` ; do
  wait $job || let "FAIL+=1"
done
if [ $FAIL -gt 0 ] ; then
  echo "fails: $FAIL"
fi
```

A.3.2 Erweiterungen über Nicht-Primkörpern

```
#!/bin/bash
###################################################################################
# usage:
#     cPN.sh
#     cPN.sh TOP
#     cPN.sh TOP "PRIME1 PRIME2 ..."
#
# Last modified: Sun Sep 07, 2014  09:33

##################################################################################
# Config
PATHBASE="/tmp/cPNrel/"
if [ $# -eq 2 ]; then
  LIMIT=$1
  LIST=$2
elif [ $# -eq 1 ]; then
  LIST="2" # 3 5 7 11 13 17"
  LIMIT=$1
else
  LIST="2" # 3 5 7 11 13 17"
  LIMIT=3
fi
LISTm="2 3 4"

nCPUm1=3

##################################################################################
FAIL=0
mkdir -p $PATHBASE
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
mkdir -p "${DIR}/cPNout/"
SRC="${DIR}/../src/"

genPathHS(){
  echo "${SRC}cPN${1}m${2}.hs"
}
genPathEx(){
  echo "${PATHBASE}cPN${1}m${2}"
}
genPathCSV(){
  echo "${DIR}/cPNout/CalcPrimNubersRel_p=${1}_m=${2}.csv"
}

genHS(){
  # genHS CHARAKTERISTIK BEGINN END
cat <<HS
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE BangPatterns #-}
module Main
  where
import System.Environment
```

```haskell
import System.Directory
import Control.Monad
import Data.Time

import GalFld.GalFld
import GalFld.More.SpecialPolys

p = ${1}
λ$(genPrimeField ${1} "PF")
outFile = "$(genPathCSV $1 $2)"

m = ${2}

pf = 1::PF

data T = T { ext :: Int -- Grad des Grundkörpers über dem Primkörper
           , deg :: Int -- Grad der Erweiterung
           , countP :: Int -- Anzahl primitiver Elemente
           , countN :: Int -- Anzahl normaler Elemente
           , countPN :: Int } -- Anzahl primitiv-normaler Elemente

genPrimNorm m n = (record, fac)
  where one    = extendFFBy m pf
        cyP    = cyclotomicPoly (p^(n*m)-1) one
        piP    = piPoly $ pTupUnsave [(n,one),(0,-1)]
        ggT    = ggTP cyP piP
        fac    = factorP ggT
        record = T m n (uDegP cyP) (uDegP piP) (uDegP ggT)

main = do
  args <- getArgs
  let indxs = [(read $ head args)..(read $ head $ tail args)]
  bool <- doesFileExist outFile
  unless bool $ putHeaderToFile p
  mapM_ (\n -> do
    let t = fst $ genPrimNorm m n
    putInfo t p
    getCurrentTime >>= print
    putToFile t p) indxs
      where putInfo !(T m n cP cN cPN) p = do
              putStrLn $ "In (F" ++ show p ++ "^" ++ show m ++ ")^" ++ show n
                ++ " über F" ++ show p ++ " gibt es:"
                ++ "\t" ++ show cP
                ++ "\t" ++ show cN
                ++ "\t" ++ show cPN
            putHeaderToFile p =
              writeFile outFile "p,m,n,#primitive,#normal,#primitivNormal\n"
            putToFile (T m n cP cN cPN) p =
              appendFile outFile $ show p ++ ","
                                    ++ show m ++ ","
                                    ++ show n ++ ","
                                    ++ show cP ++ ","
                                    ++ show cN ++ ","
                                    ++ show cPN ++ "\n"
```

```
HS
}

getDoneNumber(){
  csv=$(genPathCSV $1 $2)
  if [ -f $csv ]; then
    echo `expr $(tail -n1 $csv | cut -d',' -f2) + 1`
  else
    echo "1"
  fi
}

################################################################################
# gen and compile HS
pushd $SRC

for m in $LISTm ; do
  for p in $LIST; do
    if [ ! -f $(genPathEx $p $m) ]; then
      genHS $p $m >$(genPathHS $p $m)
      mkdir -p "${PATHBASE}compileOut/"
      ghc -j λ
        -outputdir "${PATHBASE}compileOut/" λ
        -o "$(genPathEx $p $m)" λ
        -O2 -isrc -threaded λ
        "$(genPathHS $p $m)"
      rm $(genPathHS $p $m)
    fi
  done
done
popd

################################################################################
for m in $LISTm ; do
  for p in $LIST; do
    if [ -f $(genPathEx $p $m) ]; then
      #echo $(getDoneNumber $p $m)
      $(genPathEx $p $m) $(getDoneNumber $p $m) $LIMIT &
    fi

    # limit subprocesses to 4
    while [ $(jobs -p | wc -l) -gt $nCPUm1 ] ; do
      sleep 1
    done
  done
done

################################################################################
for job in `jobs -p` ; do
  wait $job || let "FAIL+=1"
done
if [ $FAIL -gt 0 ] ; then
  echo "fails: $FAIL"
fi
```

Literatur

[1] M. Bavarian. „Lecture 6“. In: M. Sudan. *6.S897 Algebra and Computation.* Vorlesungsskript. 2012. URL: `http://people.csail.mit.edu/madhu/ST12/scribe/lect06.pdf`.

[2] J. Cohen. *Computer algebra and symbolic computation: mathematical methods.* Ak Peters Series. AK Peters, 2003.

[3] K. Geddes, S. Czapor und G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992. URL: `http://books.google.de/books?id=9f0UwkkRxT4C`.

[4] D. Hachenberger. *Endliche Körper I.* Vorlesungsskript. Universität Augsburg, SS 2013.

[5] D. Hachenberger. *Endliche Körper II.* Vorlesungsskript. Universität Augsburg, WS 2013/2014.

[6] D. Hachenberger. „On Primitive and Free Roots in a Finite Field.“ In: *Appl. Algebra Eng. Commun. Comput.* 3 (1992), S. 139–150.

[7] HaskellWiki, Hrsg. *Monad.* URL: `http://www.haskell.org/haskellwiki/Monad`.

[8] HaskellWiki, Hrsg. *Performance.* URL: `http://www.haskell.org/haskellwiki/Performance`.

[9] HaskellWiki, Hrsg. *Thunk.* URL: `http://www.haskell.org/haskellwiki/Thunk`.

[10] G. Hutton. *Programming in Haskell.* Cambridge University Press, Jan. 2007.

[11] M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide.* 1st. San Francisco, CA, USA: No Starch Press, 2011. URL: `http://learnyouahaskell.com`.

[12] S. Marlow u. a. *Haskell 2010 Language Report.* 2010. URL: `http://www.haskell.org/haskellwiki/Language_and_library_specification`.

[13] M. M. Maza. „From Newton to Hensel“. In: *Foundations of Computer Algebra.* Vorlesungsskript. 2008. URL: `http://www.csd.uwo.ca/~moreno/CS424/Lectures/FastDivisionAndGcd.ps.gz`.

[14] S. Peyton Jones u. a. „The Haskell 98 Language and Libraries: The Revised Report“. In: *Journal of Functional Programming* 13.1 (Jan. 2003). `http://www.haskell.org/definition/`, S. –255.

[15] M. Rabin. *Probabilistic Algorithms in Finite Fields.* Technical report. Massachusetts Inst. of Technology, Lab. for Computer Science, 1979. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-213.pdf.

[16] T. Sauer. *Computeralgebra.* Vorlesungsskript. 2010. URL: https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/sauer/geyer/ComputerAlgebra.pdf.

[17] Z. Wan. *Lectures on Finite Fields and Galois Rings.* World Scientific, 2003.

[18] Wikipedia. *Horner-Schema — Wikipedia, Die freie Enzyklopädie.* 2014. URL: http://de.wikipedia.org/w/index.php?title=Horner-Schema&oldid=130454488.

[19] Wikipedia. *Kernel (linear algebra).* 2014. URL: http://en.wikipedia.org/wiki/Kernel_(linear_algebra).

[20] Wikipedia. *Synthetic division — Wikipedia, The Free Encyclopedia.* 2014. URL: http://en.wikipedia.org/w/index.php?title=Synthetic_division&oldid=610743729.

https://github.com/maximilianhuber/softwareProjekt/