

Inhaltsverzeichnis

1	Haskell	1
1.1	Über die Programmiersprache	1
1.2	Ausführen von Haskell Programmen	2
1.3	Installieren von Haskell Paketen	3
1.4	Entwicklung von Haskell-Code	3
1.4.1	Testing: <code>hspec</code>	3
1.4.2	Benchmarking: <code>criterion</code>	4
1.4.3	Zusammenfügen: <code>cabal</code>	4
1.4.4	Dokumentation: <code>haddock</code>	4
1.5	Das Haskell Typensystem	5
1.6	Pragmas	5
2	Implementierung	7
2.1	Implementierung von Polynomen	8
2.1.1	Der Datentyp	8
2.1.2	Instanzen	9
2.1.3	Polynome erstellen	10
2.1.4	Einwertige Operationen auf Polynomen	11
2.1.5	Zweiwertige Operationen auf Polynomen	13
2.1.6	Weiteres	16
2.2	Alternative Polynomalgorithmen	17
2.2.1	Verschiedene Multiplikationsalgorithmen	17
2.2.2	Division mit Rest mit Inversen $\text{mod } x^l$	24
2.3	Endliche Körper	28
2.3.1	Primkörper	28
2.3.2	Erweiterungskörper	31
2.4	Lineare Algebra	33
2.4.1	Erzeugung von Matrizen und Basisoperationen	33
2.4.2	Zweiwertige Operationen auf Matrizen	36
2.4.3	Lineare Algebra	37
2.4.4	Weiteres	40
2.5	Faktorisierung von Polynomen über endlichen Körpern	40
2.5.1	Triviale Faktoren	40
2.5.2	Funktionen rund um Faktorisierungen	41
2.6	Weiteres	42
2.6.1	Die Klasse <code>ShowTex</code>	42
2.6.2	Spezielle Polynome und zahlentheoretische Funktionen	43

3	Algorithmen auf Polynomen über endlichen Körpern	47
3.1	Quadratfreie Faktorisierung	47
3.1.1	Algorithmus zur quadratfreien Faktorisierung über endlichen Körpern . .	48
3.2	Der Algorithmus von Berlekamp	50
3.2.1	Idee	50
3.2.2	Die Berlekamp-Algebra	51
3.2.3	Der Berlekamp-Algorithmus	52
3.2.4	Alternative Implementierungen	55
3.3	Irreduzibilitätstest nach Rabin	56
3.3.1	Ein kleiner Vergleich	59
4	Beispiel: 1	61
5	Beispiel: Primitiv-normale-Elemente	62
	Anhang	64

1 Haskell



<http://xkcd.com/1312/>

1.1 Über die Programmiersprache

Die Einleitung in [8] sagt folgendes über Haskell.

- Haskell ist eine *rein funktionale* Programmiersprache. In einer *imperativen* Programmiersprache gibt man dem Computer eine Folge von Aufgaben, welche dann ausgeführt werden. Dazu gibt es Strukturen, die den Ablauf steuern, wie beispielsweise **for** und **while**.

Anders hingegen in einer funktionalen Programmiersprache. Man sagt dem Computer nicht, was er tun soll. Stattdessen sagt man ihm eher, was die Sachen sind. Zum Beispiel, kann man dem Computer sagen, dass die Fakultät einer Zahl das Produkt aller Zahlen von 1 bis zu dieser Zahl ist. Dies wird als eine, häufig rekursive, Funktion ausgedrückt.

Beim funktionalen Programmieren kann man keine Werte von Variablen verändern. In einer rein funktionalen Programmiersprache hat eine Funktion keine Seiteneffekte. Das einzige was eine Funktion tun kann, ist eine Berechnung, basierend auf ihren Eingaben. Das scheint eine Einschränkung zu sein, aber in der Tat hat dies einige positive Konsequenzen. Beispielsweise liefert eine Funktion bei gleichen Eingaben, unabhängig von der Umgebung, immer den gleichen Rückgabe Wert. Diese Eigenschaft heißt *Referentielle Transparenz*.

- Haskell ist *lazy*. Das bedeutet, dass Haskell Funktionen nicht auswertet, solange das Ergebnis nicht benötigt wird. Dies wird durch Referentielle Transparenz ermöglicht. Haskell bemüht sich, die Auswertungen von Ausdrücken so lange wie möglich zu vermeiden. Es wird damit auch ermöglicht scheinbar unendliche Datenstrukturen zu verwenden, da nur Teile, also so weit wie nötig, evaluiert werden.
- Haskell ist *statisch Typisiert*. Das bedeutet, dass der Computer bereits zur Compilezeit weiß, welcher Teil eine Zahl ist und was eine Funktion ist, die aus einer Liste von Zahlen einen String macht, usw. Das bedeutet, dass viele Fehler bereits während des Compilierens erkannt werden können.

Zusätzlich ist Haskell auch noch sehr gut darin, Typen zu inferieren. Das bedeutet, dass man meist nicht extra angeben muss, welchen Typ jeder Teil im Code hat. Beispielsweise erkennt Haskell aus `a = 4 + 5` dass `a` eine Zahl sein muss. Damit ist es auch leichter, allgemeineren Code zu schreiben, der an vielen Stellen anwendbar ist.

- Haskell ist *elegant und präzise*. Da Haskell viele Konzepte höherer Programmiersprachen nutzt, sind in Haskell geschriebene Programme meist kürzer als ein vergleichbares imperatives. Und kürzere Programme sind einfacher zu warten und enthalten weniger Fehler.

Die Haskell Entwicklung begann 1987, als sich eine Gruppe von Wissenschaftlern zusammengesetzt hat, um eine Programmiersprache zu entwickeln, die ihren Ansprüchen genügt. Der *Haskell Report*, welcher die erste stabile Version beschreibt, wurde 1999 publiziert (überarbeitete Version: [11]). Der aktuelle Standard wird beschrieben in [9].

Gute Bücher zum Einstieg in Haskell sind beispielsweise [7] und [8]. Basierend auf [7] gibt es von Erik Meijer auch eine ausführliche Video Reihe, die leicht im Internet zu finden ist. Zum Buch [8] gibt es ebenfalls im Internet eine vollständige HTML Variante¹.

Eine ausführliche Übersicht über Tutorials bietet die Seite <http://www.haskell.org/haskellwiki/Tutorials>.

Eine Liste an Büchern bietet <http://www.haskell.org/haskellwiki/Books>.

1.2 Ausführen von Haskell Programmen

Haskell kann jederzeit interpretiert oder kompiliert werden. Mit dem Interpreter `ghci` oder `hugs` kann man einfach Programme oder Code-Schnipsel testen. Alternativ erhält man durch kompilieren mit `ghc` ausführbare Dateien, welche dank recht umfangreicher Optimierung performanter sind. Für eine ausführlichere Optimierung gibt es den Compiler Parameter `-O`. Noch mehr Optimierung verspricht `-O2`, wobei das kompilieren damit nochmals deutlich länger dauert.

Die Compileroption `-threaded` bereitet die ausführbare Datei darauf vor, parallel ausgeführt zu werden. Zusätzlich muss man beim ausführen dann noch die Parameter `-RTS -N4` mitgeben, wobei die 4 die Anzahl der Prozessorkerne angibt, die genutzt werden sollen.

¹<http://learnyouahaskell.com/>

1.3 Installieren von Haskell Paketen

Zum installieren gibt es das Konsolenwerkzeug `cabal`² für Windows und Unix Systeme. Durch ausführen von

```
cabal update
```

holt sich dieses, aktuelle Paketlisten von <https://hackage.haskell.org/>. Danach kann man mittels

```
cabal install PAKETNAME
```

Pakete aus der umfangreichen Bibliothek installieren. Dabei löst `cabal` selbstständig die Abhängigkeiten auf.

1.4 Entwicklung von Haskell-Code

Für Haskell gibt es eine umfangreiche Auswahl an Programmen, die einem bei der Entwicklung von Haskell Bibliotheken und Programmen helfen. Hier sollen die erwähnt werden, die für dieses Projekt genutzt wurden.

1.4.1 Testing: hspect

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.³

Hspec ermöglicht es einfach tests zu schreiben, deren Quellcode leicht verständlich ist und eine konkrete Aussage darüber trifft, was die getestete Funktion tun sollte.

Ein einfaches und selbsterklärendes Beispiel⁴ 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 $ \x xs → head (x:xs) ≡ (x :: Int)

    it "throws an exception if used with an empty list" $ do
```

²<http://www.haskell.org/cabal/download.html>

³<https://hackage.haskell.org/package/hspec>

⁴<http://hspec.github.io/>

```
evaluate (head []) `shouldThrow` anyException
```

Ein Ausführen, durch beispielsweise `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.⁵

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.⁶

Weitere Quellen:

- http://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program

1.4.4 Dokumentation: haddock

Haddock is a tool for automatically generating documentation from annotated Haskell source code.⁷

⁵<https://hackage.haskell.org/package/criterion>

⁶<https://hackage.haskell.org/package/Cabal>

⁷<http://www.haskell.org/haddock/>

1.5 Das Haskell Typensystem

In Haskell weiß jeder Ausdruck bereits zur Compilezeit, welchen Typ er hat. So werden Fehler wie das Dividieren eines Boolean durch eine Zahl bereits während des Compilierens aufgedeckt und müssen nicht durch Glück während des Ausführens gefunden werden. Gekennzeichnet werden Typenangaben in Haskell durch `::` als Infix-Operator. Die folgende Aussage bedeutet, dass eine Variable `count` vom Typ `Int` ist.

```
count :: Int
```

Da es in Haskell Higher-Order Funktionen gibt haben natürlich auch Funktionen einen Typ. Beispielsweise hat die Funktion `head`, welche zu einer Liste das erste Element wiedergibt, den Typ:

```
head :: [a] → a
```

Zu bemerken ist hier, dass für die Funktion nicht vorgegeben ist, von welchem Typ die Elemente der Liste sind. So kann die Typen Variable `a` für jeden Typen stehen, also funktioniert die Funktion beispielsweise auf Listen von Zahlen. Ebenso funktioniert sie aber für Strings, welche in Haskell als Listen von Zeichen implementiert sind.

Weiter gibt es Typen Klassen, welche beispielsweise den Interfaces von Java ähneln. Eine Klasse beschreibt dabei ein Verhalten von Typen. Um zu sagen, dass ein Typ das Verhalten einer Klasse hat, muss man für diesen eine Instanz dieser Klasse implementieren. Dazu besteht jede Klasse aus einer vorgegebenen Liste von Funktionen, welche implementiert werden müssen. Ein gutes Beispiel hierfür ist `Eq`, welches als einzige Funktion den Operator `(=)`⁸ enthält. Also muss man sagen, welche Werte oder Konstruktorterme des Typs gleich sein sollen.

Die Funktion `(=)` selbst hat den Typ

```
(=) :: (Eq a) ⇒ a → a → Bool
```

wobei das `⇒` ein Zeichen ist, das eine Typen Klassen Restriktion beschreibt. Hier darf also die Typen Variable `a` nicht durch alle Typen ersetzt werden, sondern nur durch die, die eine Instanz `Eq` haben. Also können wir Werte auf Gleichheit nur dann Prüfen, wenn wir eine Instanz `Eq` haben. Freie Datentypen werden, sofern nicht extra eine Instanz von `Eq` erzeugt wurde, auf strukturelle Gleichheit geprüft.

Eine umfangreiche Liste an wichtigen Typen Klassen sowie eine ausführlichere Erklärung des Haskell Typensystems findet man beispielsweise in [8] im zweiten Kapitel.

1.6 Pragmas

Pragmas⁹ bieten in Haskell die Möglichkeit, für den Compiler bestimmte Commandos in den Quellcode zu integrieren. Diese beeinflussen meist nicht die Bedeutung des Codes sondern haben eher Einfluss auf Effizienz des generierten Programms. Auch können damit Haskell Erweiterungen aktiviert werden.

Ein (Sprach) Pragma im Code ist berandet durch `{-#... #-}` und ein Beispiel dafür wäre

⁸Die Schreibweise mit den Klammern dient dazu, Infix-Operatoren zu definieren.

⁹https://www.haskell.org/ghc/docs/7.0.4/html/users_guide/pragmas.html


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

Dadurch werden die (Sprach-) Erweiterungen `CPP` und `TemplateHaskell` aktiviert. Die erste Erweiterung ermöglicht es durch die Befehle `#if 1` bzw. `#if 0`, `#else` und `#endif` analog zu `/iftrue` und `/iffalse` im \LaTeX mehrere Zeilen im Quellcode zu deaktivieren bzw. schnell zwischen alternativen Implementierungen umzuschalten.

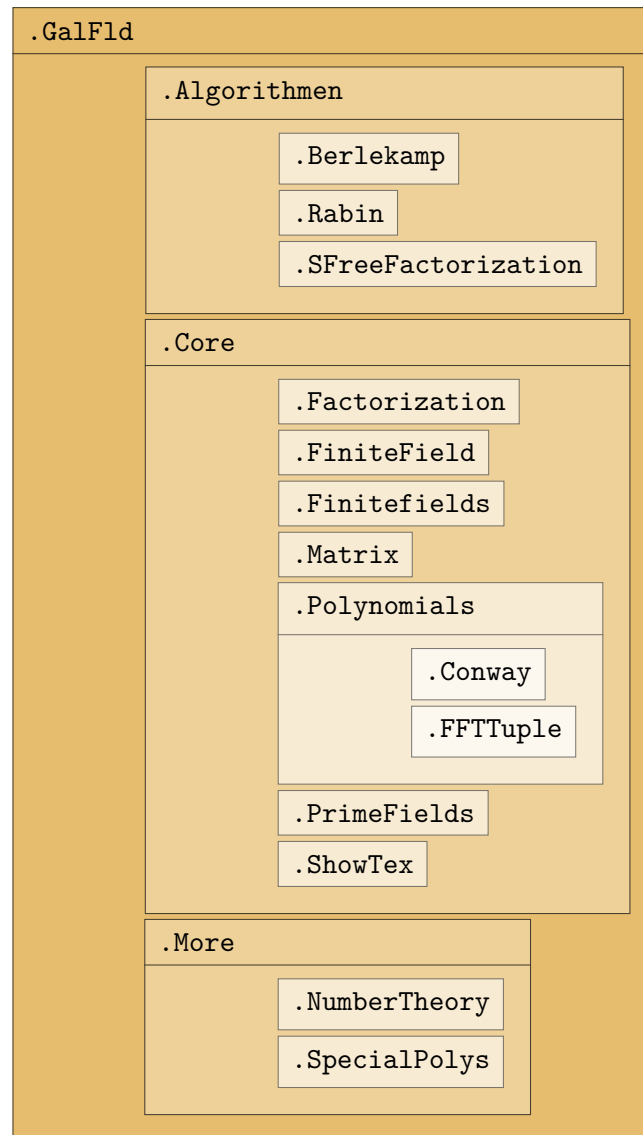
Die `TemplateHaskell` ermöglicht es, zusammen mit `QuasiQuotes` Haskell Funktionen bereits zur Compilezeit auszuführen. Damit lässt sich dynamisch Code erzeugen oder man kann auch Rechenaufgaben in die Compilezeit verlagern.

Es gibt noch diverse andere Arten von Pragmas, die beispielsweise:

- `OPTIONS_GHC` Pragmas bieten eine Möglichkeit, dem Compiler direkt Compileparameter zu übergeben und
- `INLINE` Pragmas werden in der Form `{-#INLINE funktionsname #-}` einer Funktion direkt vorangestellt 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 `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 $f(X) = a_n X^n + \dots + a_0$ als Liste¹ der Länge n zu hinterlegen,
- oder man speichert lediglich eine Liste von Tupel (i, a_i) , so dass $i \in \{0, \dots, n\}$ 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:

```
31 -- Polynome sind Listen von Monomen, welche durch Paare (Integer,a)
32 -- dargestellt werden. In der ersten Stelle steht der Grad, in der zweiten der
33 -- Koeffizient.
34 data Polynom a = PMS { unPMS :: [(Int,a)], clean :: Bool } deriving ()
```

GalFld/
[.hs]

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

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

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

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/
[.hs]

¹Was genau eine "Liste" in der jeweilig benutzten Sprache bedeuten soll, bleibt der Interpretation überlassen.

```

36 -- |Lösche (i,0) Paare und sortiere dem Grade nach absteigend
37 cleanP :: (Num a, Eq a) => Polynom a -> Polynom a
38 cleanP f@(PMS ms True) = f
39 cleanP (PMS ms False) = PMS (clean' ms) True
40   where clean' ms = filter (\(_,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 zu 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.

```

41 Eq instance (Eq a, Num a) => Eq (Polynom a) where
42   f ≡ g = eqP f g

```

GalFld/
[.hs]

```

43 Num instance (Num a, Eq a) => Num (Polynom a) where
44   {-# INLINE (+) #-}
45   f@(PMS _ _) + g@(PMS _ _) = PMS hs True
46     where hs = addPM (unPMS $ cleanP f) (unPMS $ cleanP g)
47   {-# INLINE (-) #-}
48   f@(PMS _ _) - g@(PMS _ _) = PMS hs True
49     where hs = subtrPM (unPMS $ cleanP f) (unPMS $ cleanP g)
50   {-# INLINE (*) #-}
51   f@(PMS _ _) * g@(PMS _ _) = PMS hs True
52     where hs = multPM (unPMS $ cleanP f) (unPMS $ cleanP g)
53   fromInteger i      = PMS [(0,fromInteger i)] True
54   abs _              = error "Prelude.Num.abs: inappropriate abstraction"
55   signum _           = error "Prelude.Num.signum: inappropriate abstraction"
56   negate (PMS ms b) = PMS ((map . A.second) negate ms) b

```

GalFld/
[.hs]

```

57 Show instance (Show a, Eq a, Num a) => Show (Polynom a) where
58   show (PMS [] _) = "0"
59   show (PMS ms True) = show' $ tuple2List ms
60     where show' ms = intercalate "+" $
61       (λss -> [s | s <- reverse ss, s ≠ ""]) $
62         zipWith (curry show') ms [0..]
63   show' :: (Show a, Eq a, Num a) => (a,Int) -> String
64   show' (0,_) = ""
65   show' (m,0) = show m
66   {-show' (1,i) = showExp i-}
67   show' (m,i) = show m # "." # showExp i
68   showExp :: Int -> String
69   showExp 0 = ""
70   showExp 1 = "\x1B[04mX\x1B[24m"
71   showExp i = "\x1B[04mX" # showExp' (show i) # "\x1B[24m"
72   showExp' :: String -> String

```

GalFld/
[.hs]

```

73     showExp' "" = []
74     showExp' (c:cs) = newC : showExp' cs
75     where newC | c == '0' = '0'
76                 | c == '1' = '1'
77                 | c == '2' = '2'
78                 | c == '3' = '3'
79                 | c == '4' = '4'
80                 | c == '5' = '5'
81                 | c == '6' = '6'
82                 | c == '7' = '7'
83                 | c == '8' = '8'
84                 | c == '9' = '9'
85     show f = show $ cleanP f

```

```

ShowTex
86 instance (ShowTex a, Num a, Eq a) => ShowTex (Polynom a) where
87     showTex (PMS [] _) = "0"
88     showTex (PMS ms True) = show' $ tuple2List ms
89     where show' ms = intercalate "+" $
90         (\ss -> [s | s <- reverse ss, s /= ""]) $
91         zipWith (curry showTex') ms [0..]
92     showTex' :: (ShowTex a, Eq a, Num a) => (a,Int) -> String
93     showTex' (0,_) = ""
94     showTex' (m,i) = showTex m # showExp i
95     showExp :: Int -> String
96     showExp 0 = ""
97     showExp 1 = "\\cdot{X}"
98     showExp i = "\\cdot{X}^{ " # show i # "}"
99     showTex f = showTex $ cleanP f

```

GalFld/
[.hs]

2.1.3 Polynome erstellen

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

Diese sind selbsterklärend:

```

100 -- |Erzeuge ein Polynom aus einer Liste von Koeffizienten
101 pList :: (Num a, Eq a) => [a] -> Polynom a
102 pList ms = PMS (list2TupleSave ms) True

```

GalFld/
[.hs]

```

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

```

GalFld/
[.hs]

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

```

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

```

GalFld/
[.hs]

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

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

GalF1d/
[.hs]

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

GalF1d/
[.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:

```
115 -- |Das Nullpolynom
116 nullP = PMS [] True
```

GalF1d/
[.hs]

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

GalF1d/
[.hs]

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

```
122 -- |Erzeugt ein konstantes Polynom, d.h. ein Polynom von Grad 0
123 pK Konst :: (Eq a, Num a) => a -> Polynom a
124 pK Konst x | x == 0       = nullP
125                | otherwise = PMS [(0,x)] True
```

GalF1d/
[.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:

```
126 {-# INLINE degP #-}
127 -- |Gibt zu einem Polynom den Grad
128 degP :: (Num a, Eq a) => Polynom a -> Maybe Int
129 degP f@(PMS [] _)    = Nothing
130 degP (PMS ms True)    = Just $ fst $ head ms
131 degP f                = degP $ cleanP f
```

GalF1d/
[.hs]

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

```
132 {-# INLINE uDegP #-}
133 uDegP :: (Num a, Eq a) => Polynom a -> Int
134 uDegP = fromJust . degP
```

GalF1d/
[.hs]

Auswerten Natürlich muss auch etwas in ein Polynom einsetzen können, was wir mit Hilfe des Hornerschemas (vgl. [14]) implementiert haben. Dies zeigt eine schöne Anwendung der Haskell-

```

135 {-# INLINE evalP #-}
136 -- [Nimmt einen Wert und ein Polynom und wertet das Polynom an dem Wert aus.
137 -- Mittels Horner Schema
138 evalP :: (Eq a, Num a) => a -> Polynom a -> a
139 evalP x f = evalP' x (unPMS $ cleanP f)
140 evalP' :: (Num a) => a -> [(Int,a)] -> a
141 evalP' x [] = 0
142 evalP' x fs = snd $ foldl' (\(i,z) (j,y) -> (j,z*x^(i-j)+y)) (head fs) (tail fs)

```

GalFld/
[.hs]

Normieren Über das Normieren braucht man nicht viele Worte verlieren.

```

143 moniP :: (Num a, Eq a, Fractional a) => Polynom a -> Polynom a
144 moniP f@(PMS [] _) = f
145 moniP f@(PMS ms True) = PMS ns True
146   where ns = map (\(i,m) -> (i,m/l)) ms
147         l = snd $ head ms
148 moniP f = moniP $ cleanP f

```

GalFld/
[.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:

```

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

```

GalFld/
[.hs]

Formale Ableitung

```

156 -- [Nimmt ein Polynom und leitet dieses ab.]
157 deriveP :: (Num a, Eq a) => Polynom a -> Polynom a
158 deriveP (PMS [] _) = PMS [] True
159 deriveP (PMS ms b) = PMS (deriveP' ms) b
160   where deriveP' [] = []
161         deriveP' ((i,m):ms) | j<0 = deriveP' ms
162                             | c=0 = deriveP' ms
163                             | otherwise = (j,c) : deriveP' ms
164         where j=i-1
165               c=m*fromInteger (fromIntegral i)

```

GalFld/
[.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

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

GalFld/
[.hs]

```

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

```

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

```

170 {-# INLINE reciprocP #-}                                     GalFld/
                                                                [.hs]
171 reciprocP :: (Eq a, Fractional a) => Polynom a -> Polynom a
172 reciprocP f = reciprocP2 d f
173     where d = uDegP f

```

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:

```

174 {-# INLINE multMonomP #-}                                     GalFld/
                                                                [.hs]
175 -- |Multipliziert f mit x^i
176 multMonomP :: (Eq a, Num a) => Int -> Polynom a -> Polynom a
177 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:

```

178 {-# INLINE eqP #-}                                           GalFld/
                                                                [.hs]
179 eqP :: (Eq a, Num a) => Polynom a -> Polynom a -> Bool
180 eqP (PMS ms True) (PMS ns True) = eqP' ms ns
181     where eqP' [] ns = isNullP' ns
182           eqP' ms [] = isNullP' ms
183           eqP' ((i,m):ms) ((j,n):ns) = i==j && m==n && eqP' ms ns
184 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:

```

185 {-# INLINE addPM #-}                                         GalFld/
                                                                [.hs]
186 -- | addiere Polynome in Monomdarstellung, d.h
187 -- [(Int,a)] wobei die Liste in Int ABSTEIGEND sortiert ist
188 addPM :: (Eq a, Num a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]
189 addPM [] gs = gs
190 addPM fs [] = fs
191 addPM ff@((i,f):fs) gg@((j,g):gs)
192   | i==j && c/=0 = (i,c) : addPM fs gs
193   | i==j && c==0 = addPM fs gs
194   | i<j        = (j,g) : addPM ff gs
195   | i>j        = (i,f) : addPM fs gg
196     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) werden. Um dies zu verhindern, haben wir die Subtraktion separat geschrieben. \square

```

197 {-# INLINE subtrPM #-}
198 -- | subtrahiere Polynome in Monomdarstellung, d.h
199 -- [(Int,a)] wobei die Liste in Int ABSTEIGEND sortiert ist
200 subtrPM :: (Eq a, Num a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]
201 subtrPM [] gs = map (A.second negate) gs
202 subtrPM fs [] = fs
203 subtrPM ff@((i,f):fs) gg@((j,g):gs)
204   | i==j && c/=0 = (i,c) : subtrPM fs gs
205   | i==j && c==0 = subtrPM fs gs
206   | i<j        = (j,negate g) : subtrPM ff gs
207   | i>j        = (i,f) : subtrPM fs gg
208   where !c = f-g

```

GalF1d/
[.hs]

Wiederum darf obige Subtraktion nur auf wohlgeformte Polynome angewandt werden.

Multiplikation Da auf dem Datentyp `Int` sowohl Multiplikation als auch Addition in gleicher Zeit erfolgen, hat sich herausgestellt, dass in den meisten Fällen die „Standardmultiplikation“ die effizienteste ist. Diese ist wie folgt implementiert: \square

```

209 {-# INLINE multPM #-}
210 -- | Multiplikation von absteigend sortierten [(Int,a)] Listen
211 multPM :: (Eq a, Num a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]
212 multPM f [] = []
213 multPM [] f = []
214 multPM ms ns = foldr1 addPM summanden
215   where summanden = [multPM' i m ns | (i,m) <- ms]

```

GalF1d/
[.hs]

```

216 {-# INLINE multPM' #-}
217 multPM' i m [] = []
218 multPM' i m ((j,n):ns) | c == 0 = multPM' i m ns
219                           | otherwise = (k,c) : multPM' i m ns
220   where !c = n*m
221         !k = i+j

```

GalF1d/
[.hs]

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 am langsamsten erwiesen und wurde daher wieder aus dem Code entfernt. Die nun in den meisten Fällen am effizientesten Methode ist die Division mit Hilfe des Hornerschemas. Eine sehr gute und ausführliche Erklärung findet sich in [16].

Wiederum lässt sich die Division per Hornerschema sehr schön rekursiv in Haskell niederschreiben.

```

222 {-# INLINE divPHornerM' #-}                                GalF1d/
223 -- |Horner für absteigend sortierte [(Int,a)] Paare          [.hs]
224 divPHornerM' _ [] _ _ = []
225 divPHornerM' divs ff@((i,f):fs) lc n
226   | n > fst (head ff) = ff
227   | otherwise         = (i,fbar) : divPHornerM' divs fs lc n
228   where fbar = f/lc
229         {-# INLINE hs #-}
230         hs   = addPM fs $! js
231         {-# INLINE js #-}
232         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.

```

233 -- |divP mit Horner Schema                                GalF1d/
234 -- siehe http://en.wikipedia.org/wiki/Synthetic_division    [.hs]
235 divP :: (Show a, Eq a, Fractional a) =>
236       Polynomial a -> Polynomial a -> (Polynomial a,Polynomial a)
237 divP = divPHorner

```

```

238 divPHorner a (PMS [] _) = error "Division by zero"          GalF1d/
239 divPHorner a@(PMS as True) b@(PMS bs True)                  [.hs]
240   | isNullP a         = (PMS [] True,PMS [] True)
241   | degDiff <= 0      = (PMS [] True,a)
242   | otherwise         = toP $ A.first (map (A.first (\i -> i-degB))) $
243                               splitAt splitPoint horn
244   where horn          = divPHornerM' bs as lc degB
245         degDiff       = uDegP a - uDegP b + 1
246         bs            = tail $ unPMS $ negate b
247         as            = unPMS a
248         lc            = getLcP b
249         degB          = uDegP b
250         splitPoint    = length [i | (i,j) <- horn, i >= degB]
251         toP (a,b)     = (PMS a True, PMS b True)
252 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 anderes 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:

```

253 {-# INLINE modByP #-}                                GalF1d/
254 -- |Nimmt ein Polynom und rechnet modulo ein anderes Polynom.    [.hs]
255 -- Also Division mit rest und Rückgabewert ist der Rest.
256 --

```

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

```

257 modByP :: (Show a, Eq a, Fractional a) => Polynom a -> Polynom a -> Polynom a
258 modByP f p = snd $ divP f p

```

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

```

259 {-# INLINE eekP #-}                                     GalFld/
260 -- |Erweiterter Euklidischer Algorithmus: gibt (d,s,t) zurück mit      [.hs]
261 --   ggT(a,b) = d = s*a + t*b
262 eekP :: (Show a, Eq a, Fractional a) => Polynom a -> Polynom a
263                                     -> (Polynom a, Polynom a, Polynom a)
264 eekP f g | g == 0      = (moniP f, PMS [(0, recip $ getLcP f)] True, PMS [] True)
265               | otherwise = (d,t,s-t*q)
266   where (q,r) = divP f g
267           (d,s,t) = eekP g r

```

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

```

268 {-# INLINE ggTP #-}                                     GalFld/
269 -- |Algorithmus für ggT                                              [.hs]
270 ggTP :: (Show a, Eq a, Fractional a) => Polynom a -> Polynom a -> Polynom a
271 ggTP f g = (\(x,_,_) -> x) $ eekP f g

```

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.

```

272 hasNs :: (Eq a, Fractional a) => Polynom a -> [a] -> Bool          GalFld/
273 hasNs f es = not (null [f | e <- es, evalP e f == 0])           [.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.

```

274 getAllMonicPs :: (Num a, Fractional a, Eq a) => [a] -> [Int] -> [Polynom a] GalFld/
275 getAllMonicPs es is = map (`PMS` True) $ concat [allMonics i | i <- is]      [.hs]
276   where allMonics 0 = [[(0,1)]]
277         allMonics i = [(i,1):rs | rs <- ess (i-1)]
278         ess i       | i == 0      = [[(0,y)] | y <- swpes]
279                   | otherwise = [(i,y)] | y <- swpes # ess (i-1) #
280                               [(i,y):ys | y <- swpes, ys <- ess (i-1)]
281         swpes       = filter (/= 0) es

```

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

```

[ ]                                     GalFld/
[ ]                                     [.hs]

```

```

283 getAllMonicP :: (Num a, Fractional a, Eq a) => [a] -> Int -> [Polynom a]
284 getAllMonicP es d = getAllMonicPs es [0..d]

```

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

```

285 -- [Nimmt eine Liste und eine Liste von Gräde und erzeugt daraus alle
286 -- Polynome deren Gräde in der Liste enthalten sind]
287 getAllPs :: (Num a, Fractional a, Eq a) => [a] -> [Int] -> [Polynom a]
288 -- TODO: Man muss nur das letzte Element in der Liste verändern
289 getAllPs es ds = [PMS (map (A.second (e*)) $ unPMS f) True | f <- getAllMonicPs es ds
290                      , e <- es , e ≠ 0]

```

GalFld/

```

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

```

GalFld/

[.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 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

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

Damit können die ursprünglich 4 auftretenden Multiplikation des Standardalgorithmus durch 3 Multiplikation und 4 Additionen ersetzt werden. Dies lässt sich natürlich rekursiv anwenden.

Da die Polynome jedoch nicht immer von gleichem Grad 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 Gräde als Teilungspunkt für den rekursiven Aufruf. Die Implementierung erfolgt dabei in drei Schritten:

```

297 {-# INLINE multPK #-}                                     GalFld/
298 multPK :: (Show a, Num a, Eq a) => Polynom a -> Polynom a -> Polynom a  [.hs
299 multPK f g = PMS h True
300   where h = multPMKaratsuba ((unPMS.cleanP) f) ((unPMS.cleanP) g)

```

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

```

301 {-# INLINE multPMKaratsuba #-}                             GalFld/
302 multPMKaratsuba :: (Show a, Num a, Eq a) => [(Int,a)] -> [(Int,a)] -> [(Int,a)]  [.hs
303 multPMKaratsuba f g = multPMK' n f g
304   where n = next2Pot (max df dg) `quot` 2
305         df = if null f then 0 else fst (head f) + 1
306         dg = if null g then 0 else fst (head g) + 1

```

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.

```

307 -- Der eigentliche Karatsuba                                GalFld/
308 multPMK' :: (Show a, Num a, Eq a) => Int -> [(Int,a)] -> [(Int,a)] -> [(Int,a)]  [.hs
309 multPMK' _ _ [] = []
310 multPMK' _ [] _ = []
311 multPMK' _ [(i,x)] g = map ((+) i A.*** (*) x) g
312 multPMK' _ f [(i,x)] = map ((+) i A.*** (*) x) f
313 multPMK' 1 [(i1,x1),(i2,x2)] [(j1,y1),(j2,y2)]
314   = [(2,p1), (1,p3-p1-p2), (0,p2)]
315   where !p1 = x1*y1
316         !p2 = x2*y2
317         !p3 = (x1+x2)*(y1+y2)
318 multPMK' n f g = addPM e1 $ addPM e2 e3
319   where -- High und Low Parts
320         {-# INLINE fH' #-}
321         fH' = takeWhile (\(i,_) -> i < n) f
322         {-# INLINE fL #-}
323         fL = map (A.first (\i -> i-n)) fH'
324         {-# INLINE fL #-}
325         fL = f \ \ fH'
326         {-# INLINE gH' #-}
327         gH' = takeWhile (\(i,_) -> i < n) g
328         {-# INLINE gH #-}
329         gH = map (A.first (\i -> i-n)) gH'
330         {-# INLINE gL #-}
331         gL = g \ \ gH'
332 -- Rekursiver Karatsuba
333 {-# INLINE p1 #-}
334 p1 = multPMK' (n `quot` 2) fH gH
335 {-# INLINE p2 #-}
336 p2 = multPMK' (n `quot` 2) fL gL
337 {-# INLINE p3 #-}
338 p3 = multPMK' (n `quot` 2) (addPM fH fL) (addPM gH gL)
339 {-# INLINE e1 #-}

```

```

340     e1 = map (A.first (+ (2*n))) p1
341     {-# INLINE e2 #-}
342     e2 = map (A.first (+n)) $ subtrPM p3 (addPM p1 p2)
343     {-# INLINE e3 #-}
344     e3 = p2

```

Ein kleiner Vergleich Auf Polynomen über den `PrimeFields` bringt der Karatsuba nur wenig 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.

FFT-Multiplikation: Der Schönhagen-Strassen-Algorithmus

Eine, was die Idee angeht, weitaus komplexere Möglichkeit Polynome zu multiplizieren, ist die FFT-Multiplikation. 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 der beiden 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

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

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

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

Beweis.

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

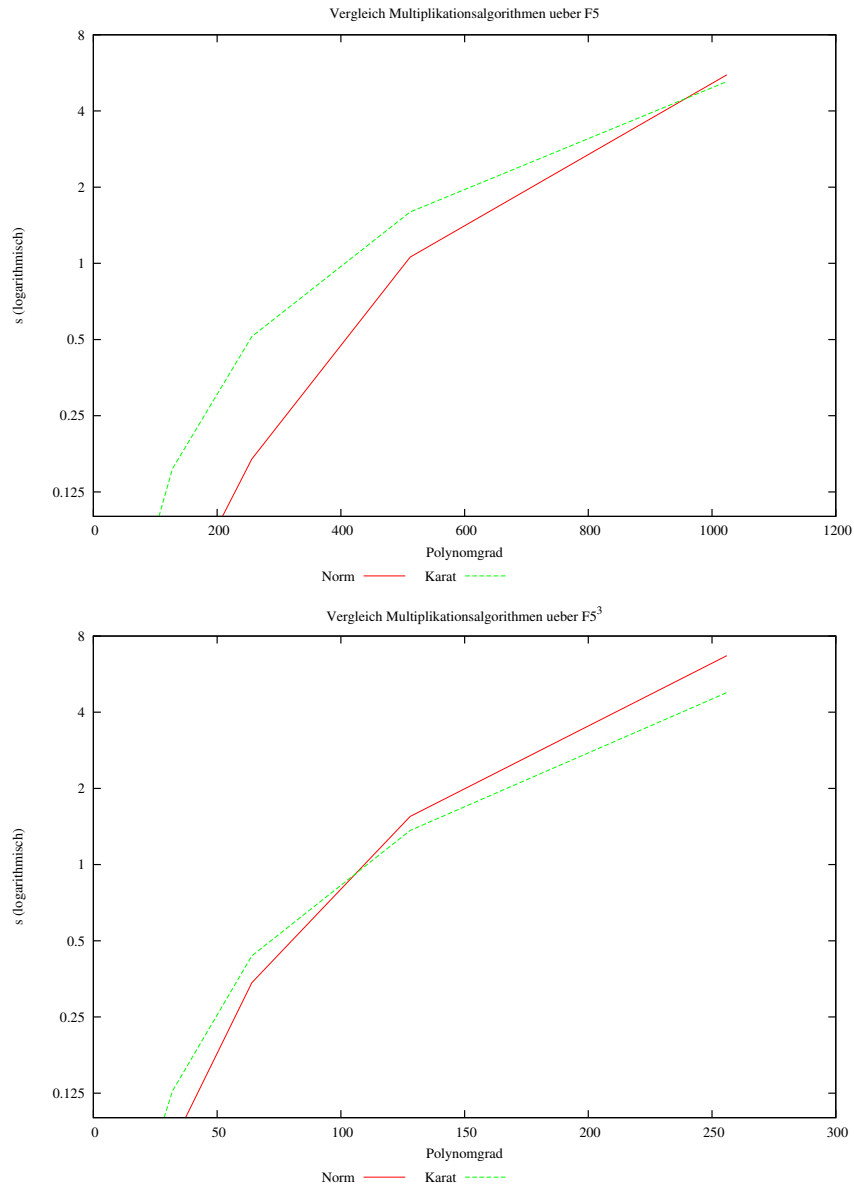
$$r(X) := \sum_{i=0}^{n-1} r_i X^i \in R[X].$$

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

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

2 Implementierung

Abbildung 2.1: Vergleich von normaler Multiplikation mit Karatsuba



2 Implementierung

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
 $r := (f_j + f_{j+\frac{n}{2}} : j = 0, \dots, \frac{n}{2})$
 $r_\omega := (\omega^j(f_j - f_{j+\frac{n}{2}}) : j = 0, \dots, \frac{n}{2})$
2. Berechne rekursiv $a := \text{FFT}(\frac{n}{2}, \omega^2, r)$ und $b := \text{FFT}(\frac{n}{2}, \omega^2, r_\omega)$
3. Mische die Ergebnisse: $f^\wedge(\omega) := (a_0, b_0, a_1, \dots, a_{\frac{n}{2}-1}, b_{\frac{n}{2}-1})$

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 es in R für jede Zweierpotenz n eine n -te Einheitswurzel und 2 muss eine Einheit sein. Die Tatsache, dass 2 eine Einheit sei, spielt für den Fall der Anwendung – nämlich Multiplikation von Polynomen über endlichen Körpern – keine Rolle, da dies dort ja immer gegeben ist. 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.

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 Schönhagen-Strassen-Algorithmus:

Algorithmus 2.3: Schönhagen-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):

2 Implementierung

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 $h(X) := H(X, X^m) \bmod X^n - 1$.

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

```

376
377 -- Helper
378 {-# INLINE intersperseL #-}
379 -- |Intersperse mit 2 Listen
380 intersperseL :: [a] -> [a] -> [a]
381 intersperseL ys [] = ys
382 intersperseL [] xs = xs
383 intersperseL (y:ys) (x:xs) = y : x : intersperseL ys xs

384 {-# INLINE zipWith' #-}
385 -- like @zipWith@ except that when the end of either list is
386 -- reached, the rest of the output is the rest of the longer input list.
387 zipWith' :: (t -> t -> t) -> t -> [t] -> [t] -> [t]
388 zipWith' _ _ xs [] = xs
389 zipWith' f t [] ys = map (f t) ys
390 zipWith' f t (x:xs) (y:ys) = (f x y) : zipWith' f t xs ys

391 {-# INLINE log2 #-}
392 -- |ineffiziente Log 2 Berechnung
393 log2 :: Int -> Int
394 log2 0 = 0
395 log2 1 = 0
396 log2 n = log2' 1 n
397   where log2' i 1 = max 0 (i-1)
398         log2' i 2 = i
399         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. □

```

400 -- |Berechnet die FFT eines Polynoms f
401 -- Benötigt eine primitive n-te Einheitswurzel,
402 -- wobei n eine 2er Potenz ist (Dies wird NICHT überprüft!)
403 -- Diese wird dargestellt als Funktion w: Int -> a -> a,
404 -- wobei f(i,x) = w^i*x für die n-te EWL w ausgewertet.

```

2 Implementierung

```

405 --
406 -- vgl Computer Algebra Algorithmus 4.11
407 fftP :: (Show a, Num a, Eq a) => (Int -> a -> a) -> Int -> Polynom a -> [a]
408 fftP w n f = fft w (+) (-) 0 1 n (p2List f)

409 -- | Int -> a -> a Implementierung der primitiven Einheitswurzel
410 -- a -> a -> a Addition auf a
411 -- a -> a -> a Subtraktion auf a
412 -- -> a Die Null
413 -- -> Int Aktuelle 2er Potenz (Starte mit 1)
414 -- -> Int FFT bis n
415 -- -> [a] Eingangsliste
416 -- -> [a] Ausgabeliste
417 fft :: (Show a) => (Int -> a -> a) -> (a->a->a) -> (a->a->a) -> a
418 -- -> Int -> Int -> [a] -> [a]
419 fft _ _ _ _ 1 fs = fs
420 fft w addF subF zero i n fs = intersperseL ls' rs'
421   where !i' = 2*i
422         ls' = fft w addF subF zero i' m ls
423         ls = take m $ zipWith' (addF) zero fs fss
424         rs' = fft w addF subF zero i' m rs
425         rs = take m $ zipWith' (w) [i | i<[0..]] $ zipWith' (subF) zero fs fss
426         fss = drop m fs
427         !m = n `quot` 2

```

GalFld/
[.hs]

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

```

428 {-# INLINE ssP #-}
429 -- | Schönhagen-Strassen für Polynome
430 ssP :: (Show a, Fractional a, Num a, Eq a) => Polynom a -> Polynom a -> Polynom a
431 ssP f g | isNullP f || isNullP g = nullP
432         | otherwise = pTup $ ss l fs gs
433   where fs = p2Tup f
434         gs = p2Tup g
435         -- || deg f*g < 2^l ||
436         l = 1 + log2 (uDegP f + uDegP g)

```

GalFld/
[.hs]

```

437 -- |Der eigentliche Schönhagen-Strassen Algorithmus
438 -- Funktioniert nur, falls 2 eine Einheit ist!
439 ss :: (Show a, Num a, Fractional a, Eq a) => Int -> [(Int,a)] -> [(Int,a)] -> [(Int,a)]
440 -- ss funktioniert nur für l>2
441 ss 1 f g = multPM f g
442 ss 2 f g = multPM f g
443 ss l f g
444   | isNullP' f || isNullP' g = []
445   | otherwise = foldr1 (addPM) $
446     reduceModxn (2^l) $ zipWith (multx (m)) [0..] hs
447   where -- << n = 2^l = m * m' >>
448         !l' = l `quot` 2
449         !m = 2^l'
450         !m' = 2^(l-l')
451         !fs = ssBuildBlocks (m*(m'-1)) m f
452         !gs = ssBuildBlocks (m*(m'-1)) m g
453         -- auf FFT vorbereiten


```

GalFld/
[.hs]

```

454 !fs' = zipWith (multx (xi)) [0..] fs
455 !gs' = zipWith (multx (xi)) [0..] gs
456 -- FFT durchführen
457 !xi    = if odd l then 1 else 2
458 !fftFs = reduceModxn (2*m) $ fft (multx (xi*2))
459                                     (addPM) (subtrPM) [] 1 m' fs'
460 !fftGs = reduceModxn (2*m) $ fft (multx (xi*2))
461                                     (addPM) (subtrPM) [] 1 m' gs'
462 -- Multiplikation der Ergebnisse und rekursiver Aufruf von ss
463 !fftHs = reduceModxn (2*m) $ zipWith (ss (l'+1)) fftFs fftGs
464 -- Inverse-FFT
465 !hs'' = reduceModxn (2*m) $ fft (multx (xi*(2*m'-2)))
466                                     (addPM) (subtrPM) [] 1 m' fftHs
467 -- * 1/m'
468 !hs'  = map (map (A.second (\x → x / (fromIntegral m')))) hs''
469 -- Rückwandlung zu H(x,y)
470 !hs    = reduceModxn (2*m) $ zipWith (multx (xi*(2*m'-1))) [0..] hs'


```

ssBuildBlocks ist dabei Schritt 3 in Algorithmus 2.3 und gegeben durch  GalFld/
[.hs]

```

471 {-# INLINE ssBuildBlocks #-}
472 ssBuildBlocks :: (Show a, Eq a, Num a) ⇒ Int → Int → [(Int,a)] → [[(Int,a)]]
473 ssBuildBlocks 0 _ fs = [fs]
474 ssBuildBlocks n m fs = (ssBuildBlocks (n - m) m ns) + [ms]
475     where ms' = filter (\(i,x) → i ≥ n) fs
476           ns  = fs \\ ms'
477           ms  = map (A.first (\i → i-n)) ms'


```

Des Weiteren ist eine schnelle Reduktion $(\text{mod } x)^{n+1}$ nötig:  GalFld/
[.hs]

```

478 {-# INLINE reduceModxn #-}
479 -- | Reduziert die innere Liste modulo x^{n+1}
480 reduceModxn :: (Show a, Num a, Eq a) ⇒ Int → [(Int,a)] → [(Int,a)]
481 reduceModxn _ [] = []
482 reduceModxn n x@(xs:xss)
483   | 1 ≥ n    = reduceModxn n $ (hs:xss)
484   | otherwise = xs : reduceModxn n xss
485     where l  = if null xs then 0 else fst $ head xs
486           fs' = filter (\(i,x) → i ≥ n) xs
487           fs  = map (\(i,x) → (i-n,negate x)) fs'
488           gs  = xs \\ fs'
489           hs  = addPM gs fs

```

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

```

490 {-# INLINE multx #-}
491 -- | Multipliziert mit x^{i*j}
492 multx :: (Num a) ⇒ Int → Int → [(Int,a)] → [(Int,a)]
493 multx _ _ [] = []
494 multx j i xs = map (A.first (\i → i+k)) xs
495     where !k = j*i

```

2.2.2 Division mit Rest mit Inversen $\text{mod } x^l$

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 [10] und [1]. Die Idee für diese schöne und zugleich schnelle Methode liefert folgende Proposition.

Proposition 2.8 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

$$h(X) f(X) \equiv 1 \pmod{X^l}$$

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.³

Beweis. Betrachte Algorithmus 2.4 und die genauere Analyse im Beweis von [10, Theorem 2]. \square

Die konkrete Antwort liefert der folgende Algorithmus.

Algorithmus 2.4: Invertieren $\text{mod } X^l$

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

Bemerkung 2.9 Man beachte, dass in Algorithmus 2.4 stets $g_i \equiv g_{i-1} \pmod{X^{2^{i-1}}}$ 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 $\text{mod } X^l$

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 $g(X) := \text{INV_MOD_MONOM}(f, l)$
 4. Berechne $q'(X) := g(X) a_l^*(X) \pmod{X^l}$
 5. Setze $q(X) := b_m \cdot q'_{n-m}(X)$ und $r(X) := a(X) - b(X)q(X)$.

Satz 2.10 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. [10, Theorem 3]. \square

³Man spricht auch von *Multiplikationszeit*, vgl. [10, Definition 2].

Implementierung

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

```

513 divPInv :: (Show a, Eq a, Fractional a) => GalFld/
514       Polynom a → Polynom a → (Polynom a, Polynom a) [.hs]
515 divPInv a b
516   | isNullP a = (nullP, nullP)
517   | a ≡ b     = (pKonst 1, nullP)
518   | l ≤ 0     = (nullP, a)
519   | otherwise = (q', r)
520   where n = uDegP a
521         m = uDegP b
522         l = n-m+1
523         (lc, b') = moniLcP b
524         f = reciprocP2 m b'
525         g = invModMonom f l
526         q = multPInter l 0 g $ reciprocP2 n a
527         q' = multKonstP lc $ reciprocP2 (l-1) q
528         r = a - b*q'

```

`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 $\text{mod} X^l$ in Schritt 4 von Algorithmus 2.5.

```

529 {-# INLINE multPInter #-} GalFld/
530 -- |Multipliziert f mit g, wobei nur Terme mit x^l für [.hs]
531 --   l > lLow und l < lHigh betrachtet werden
532 multPInter :: (Show a, Eq a, Num a) => Int → Int → Polynom a → Polynom a → Polynom a
533 multPInter _ _ (PMS [] _) = nullP
534 multPInter _ _ _ (PMS [] _) = nullP
535 multPInter lHigh lLow f g
536   = PMS (multPMInter lHigh lLow ((unPMS.cleanP) f) ((unPMS.cleanP) g)) True

```

```

537 {-# INLINE multPMInter #-} GalFld/
538 -- |Multipliziert f mit g, wobei nur Terme mit x^l für [.hs]
539 --   l > lLow und l < lHigh betrachtet werden
540 multPMInter :: (Show a, Eq a, Num a) => Int → Int →
541   [(Int,a)] → [(Int,a)] → [(Int,a)]
542 multPMInter _ _ f [] = []
543 multPMInter _ _ [] f = []
544 multPMInter lHigh lLow ms ns = foldr1 addPM summanden
545   where summanden = [multPMInter' i m ns | (i,m) ← ms]
546   {-# INLINE multPMInter' #-}
547   multPMInter' i m [] = []
548   multPMInter' i m ((j,n):ns)
549     | k < lLow || k ≥ lHigh || c ≡ 0 = multPMInter' i m ns
550     | otherwise                      = (k,c) : multPMInter' i m ns
551   where !c = n*m
552         !k = i+j

```

Letztlich bleibt dann noch die Angabe des eigentlichen Invertierens $\text{mod } X^l$.

```

553 invModMonom :: (Show a, Num a, Eq a, Fractional a) => Polynom a -> Int -> Polynom a GalFld/
554 invModMonom h k | isNullP h = nullP
555 | otherwise = PMS (invModMonom' [(0,1)] 1) True
556 where hs = unPMS $ cleanP h
557       invModMonom' !a !l
558       | l >= k = a
559       | otherwise = --trace ("invModMonom' l="++show l++" lnew="++show lnew
560                             -- ++"\n\t=> a'="++show (pTup a')++" b="++show (pTup b)) $
561                     invModMonom' b lnew
562       where -- g_i+1 = (2*g_i - h*g_i^2) mod x^(2^i)
563             b = map (A.second negate) a' # a
564             -- a' = h*g_i^2
565             a' = multPInter lnew l hs $ multPInter lnew 0 a a
566             -- nächster Schritt
567             lnew = 2*l

```

`multPInter lnew l` stellt – wie in Bemerkung 2.9 erwähnt – sicher, dass man nur die Multiplikationen durchführt, die auch wirklich notwendig sind.

Dazu ein kleines Beispiel.

Beispiel 2.11 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

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

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 $\text{mod } X^l$ für $l = n - m + 1 = 3$, also Algorithmus 2.5, beginnen:

Setze $g_0 := 1$, $r := \lceil \log_2(3) \rceil = 2$.

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

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 2.9)! Wie man in der vorliegenden Implementierung erkennt, wurde genau dies ausgenutzt und die Rechnung sieht in diesem Fall wie folgt aus:

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

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

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

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 `Projekt.Core.PrimeFields` spezifiziert. Diese werden implementiert als `Int` Werte, die durch den Wrapper `Mod` noch Zusätzlich information enthalten, in welchem Primekörper sich das Element befindet.

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

Nun können wir durch

```
568 newtype Mod n = MkMod Int
```

GalFld/
[.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 bekommen gibt es die Funkti-

```
569 {-# INLINE unMod #-}
```

GalFld/
[.hs

```
570 unMod :: Mod n -> Int
```

```
571 unMod (MkMod k) = k
```

Einen Repräsentanten, der nichtnegativ aber kleiner als die Charakteristik ist, bekommt man

```
572 {-# INLINE getRepr #-}
```

```
573 getRepr :: (Numeral n) => Mod n -> Int
```

```
574 getRepr x = unMod x `mod` modulus x
```

GalFld/
[.hs

Die Instanzen von `Show` und `ShowTex` ermöglicht es, Elemente von Primkörpern als String oder als Rohes Tex darzustellen. GalFld/

```

575 instance (Numeral n, Show n) => Show (Mod n) where [.hs
576   show x = "\x1B[33m" + show (getRepr x) + "\x1B[39m" + showModulus x
577   where showModulus :: (Numeral n) => Mod n -> String
578         showModulus = showModulus' . show . modulus
579         showModulus' :: String -> String
580   #if 1
581     showModulus' "" = ""
582     showModulus' (c:cs) = newC : showModulus' cs
583     where newC | c == '0' = '0'
584               | c == '1' = '1'
585               | c == '2' = '2'
586               | c == '3' = '3'
587               | c == '4' = '4'
588               | c == '5' = '5'
589               | c == '6' = '6'
590               | c == '7' = '7'
591               | c == '8' = '8'
592               | c == '9' = '9'
593   #else
594     showModulus' s = "^{" # s # "}"
595   #endif

```

```

596 instance (Numeral n, Show n) => ShowTex (Mod n) where GalFld/
597   showTex x = show (unMod x) + "_{" # show (modulus x) + "}" [.hs

```

Da es sich hier um einen Nichtfreien Datentyp handelt, brauchen wir noch ein Extensionalitätsaxiom, das beschreibt, wann zwei Elemente gleich sein sollen: GalFld/

```

598 instance (Numeral n) => Eq (Mod n) where [.hs
599   {-# INLINE (==) #-}
600   x == y = (unMod x - unMod y) `rem` modulus x == 0

```

Außerdem wollen wir damit rechnen können und das ganze ist dann auch ein Endlicher Körper:

```

601 instance (Numeral n) => Num (Mod n) where GalFld/
602   {-# INLINE (+) #-}
603   x + y = MkMod $ unMod x + unMod y `rem` modulus x [.hs
604   {-# INLINE (*) #-}
605   x * y = MkMod $ unMod x * unMod y `rem` modulus x
606   fromInteger = MkMod . fromIntegral
607   abs x = error "Prelude.Num.abs: inappropriate abstraction"
608   signum _ = error "Prelude.Num.signum: inappropriate abstraction"
609   {-# INLINE negate #-}
610   negate = MkMod . negate . unMod

611 instance (Numeral n) => FiniteField (Mod n) where GalFld/
612   zero = MkMod 0 [.hs
613   one = MkMod 1
614   elems = const $ elems' one
615   where elems' :: (Numeral n) => Mod n -> [Mod n]
616         elems' x = map MkMod [0.. (modulus x - 1)]
617   charakteristik = modulus
618   elemCount = modulus
619   getReprP e = 0 * snd (head (p2Tup e))

```



```

620 {-# INLINE modulus #-}
621 modulus :: Numeral a => Mod a -> Int
622 modulus x = numValue $ modulus' x
623   where modulus' :: Numeral a => Mod a -> a
624         modulus' = const undefined

```

GalFld/
[.hs]

Zum bequemen invertieren haben wir auch noch eine Instanz `Fractional`

```

625 instance (Numeral n) => Fractional (Mod n) where
626   recip      = invMod
627   fromRational _ = error "Prelude.Fractional.fromRational: inappropriate abstraction"

```

GalFld/
[.hs]

Weiterhin haben wir noch die folgenden Instanzen:

```

628 -- Zur Serialisierung wird eine Instanz vom Typ Binary benötigt
629 instance (Numeral a) => Binary (Mod a) where
630   put (MkMod x) = put x
631   get          = do x <- get
632                 return $ MkMod x

```

GalFld/
[.hs]

```

633 instance (Numeral a, NFData a) => NFData (Mod a) where
634   rnf = rnf . unMod

```

GalFld/
[.hs]

Erzeugen von Primkörpern

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

```

635 -- Erzeugen von Primkörpern durch TemplateHaskell
636 genPrimeField :: Integer -> String -> Q [Dec]
637 genPrimeField p pfName = do
638   d <- dataD (cxt []) (mkName mName) [] [] []
639   i1 <- instanceD (cxt [])
640     (appT (conT 'Numeral) (conT (mkName mName)))
641     [funD (mkName "numValue")
642       [clause [varP $ mkName "x"]
643         (normalB $ litE $ IntegerL p) [] ] ]
644   i2 <- instanceD (cxt [])
645     (appT (conT 'Show) (conT (mkName mName)))
646     [funD (mkName "show")
647       [clause [] ( normalB $ appsE [varE (mkName "show")] ) [] ] ]
648   i3 <- instanceD (cxt [])
649     (appT (conT 'NFData) (conT (mkName mName))) []
650   t <- tySynD (mkName pfName) []
651     (appT (conT 'Mod) (conT $ mkName mName))
652   return [d,i1,i2,t]
653   where mName = 'M' : show p
654 -- ppQ x = putStrLn =<< runQ ((show . ppr) `fmap` x)

```

GalFld/
[.hs]

Da es sich hierbei um eine Funktion handelt, die per TemplateHaskell zur Compilezeit ausgeführt wird, sind zum nutzen zunächst zwei Pragmas nötig:

```

{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

```

Dann kann man sich einen Primkörper der Charakteristik 7 mit Namen PF durch die Ziele

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

erzeugt werden.

2.3.2 Erweiterungskörper

Um Erweiterungskörper darzustellen verwenden wir Polynome, welche modulo einem Minimalpolynom gelesen werden sollen. Das ganze codieren wir in dem Datentypen `FFElem`.

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

GalFld/
[.hs

Natürlich haben wir die kanonische inclusion des Grundkörpers die durch `FFKonst` realisiert ist.

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

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

Durch das Minimalpolynom x^2+x+1 erzeugen wir uns eine Grad 2 Erweiterung.

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

Hier ist `e2f2` ein erzeugendes Element in dem Erzeugtem Körper. Also reicht dieses uns, um alle Körperelemente zu bekommen. Durch eine weitere Grad 2 Erweiterung erhalten wir das folgende:

```
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·X^2+12·X+12)·X mod (12 mod ...)·X^2+(12 mod ...)·X+(12·X mod 12·X^2+12·X+12))
```

Ein Element in einer Körpererweiterung wird beispielsweise dargestellt als

- $(12 \cdot X \bmod 12 \cdot X^2 + 12 \cdot X + 12)$, welches die Äquivalenzklasse von x in $\mathbb{F}_2[x]/(x^2 + x + 1)$ bezeichnet. Die \LaTeX Darstellung dazu ist $\left(\frac{12 \cdot X}{\bmod 12 \cdot X^2 + 12 \cdot X + 12}\right)$.

- $(1_2 \bmod \dots)$ 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.

Dadurch, dass wir eine `ShowTex` Instanz haben, können wir aus `e2e2f2` auch eine $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ Darstellung erzeugen:

$$\left(\frac{(1_2 \cdot X \bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2) \cdot X}{\bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2} + \left(\frac{1_2 \cdot X \bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2}{\bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2} \right) \right)$$

Ersetzen wir $(1_2 \cdot X \bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2)$ 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 auch Berechnungen machen und erhalten beispielsweise für `e2e2f2 + e2e2f2 * e2e2f2` das folgende.

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

Was in der $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ Darstellung dem folgendem entspricht:

$$\left(\frac{(1_2 \bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2) \cdot X + \left(\frac{-1_2 \bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2}{\bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2} \right)}{\bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2} + \left(\frac{1_2 \cdot X \bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2}{\bmod 1_2 \cdot X^2 + 1_2 \cdot X + 1_2} \right) \right)$$

Funktionen auf Körpererweiterungen

Zu einem Polynom bekommen wir die Charakteristik durch die folgende Funktion. 

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

GalFld/
[.hs

Auch kann man die p -te Wurzel eines Polynomes ziehen mittels der Funktion: 

```
678 {-# INLINE charRootP #-}
679 -- |Zieht die p-te wurzel aus einem Polynom, wobei p die charakteristik ist
680 charRootP :: (Show a, FiniteField a, Num a) => Polynom a -> Polynom a
681 charRootP f | isNullP f      = --trace ("charRootP f="++show f++" => "++show nullP) $
682                               nullP
683                       | f == pKonst 1 = --trace ("charRootP f="++show f++" => "++show (pKonst 1)) $
684                               pKonst 1
685                       | otherwise    = --trace ("charRootP f="++show f++" => "++
686                               --show (pTupUnsave
687                               -- [(i,m^1) | (i,m) <- p2Tup f, i `rem` p == 0])) $
688                               pTupUnsave
689                               [(i `quot` p, m^1) | (i,m) <- p2Tup f]
690 where p = charOfP f
691       q = elemCount $ getReprP f
692       l = max (quot q p) 1
```

GalFld/
[.hs

Instanzen haben wir für `Eq`, `Show` und `ShowTex`. Zum rechnen haben wir `Num` und `Fractional`. Dazu haben wir noch `NFData` und `Binary`.

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:

```
693 -- Eine Matrix ist im inneren als ein zwei dimensionales Array dargestellt,
694 -- wobei die erste Stelle die Zeile und die zweite die Spalte darstellt
695 data Matrix a = M {unM :: Array (Int, Int) a} | Mdiag a
```

GalFld/
[.hs]

Man hätte auch die Möglichkeit gehabt 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 (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`.

```
696 {-# INLINE fromListsM #-}
697 -- |Erzeugt eine Matrix aus einer Liste von Listen von Einträgen
698 fromListsM :: [[a]] -> Matrix a
699 fromListsM [] = error "GalFld.Core.Matrix.fromListsM: empty lists"
700 fromListsM [[]] = error "GalFld.Core.Matrix.fromListsM: empty lists"
701 fromListsM ess = M $ array ((1,1),(k,1))
702                        [((i,j),ess!!(i-1)!!(j-1)) | i <- [1..k]
703                                                    , j <- [1..1]]
704     where k = length ess
705           l = length $ head ess
```

GalFld/
[.hs]

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

```
706 {-# INLINE genDiagM #-}
707 -- |Erzeugt ein Vielfaches der Einheitsmatrix
708 genDiagM :: Num a => a -> Int -> Matrix a
709 genDiagM x n = M $ array ((1,1),(n,n)) $ fillList [((i,i),x) | i <- [1..n]] n n
710     where fillList ls n m = ls * [(idx,0) | idx <- getAllIdxsExcept n m idxs]
711           where idxs      = map fst ls
712           getAllIdxsExcept n m idxs = [idx | idx <- [(i,j) | i <- [1..n]
713                                                         , j <- [1..m]]
714                                     , idx `notElem` idxs]
```

GalFld/
[.hs]

Beispiel 2.12 Möchte man die Matrix $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \in \mathbb{Z}^{2 \times 2}$ erzeugen, so gibt es die folgenden drei verschiedenen Varianten:

2 Implementierung

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 2.13 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 zurückwandeln:

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

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

```
721 -- |Gibt zu einer Matrix die Grenzen zurück                                     GalFld/
722 -- Das Ergebnis hat die Form ((1,k),(1,l))                                     [.hs
723 {-# INLINE boundsM #-}
724 boundsM :: Matrix a -> (Int,Int)
725 boundsM (M m) = snd $ bounds m
```

```
726 -- |Gibt zu einer Matrix die Anzahl der Zeilen zurück                         GalFld/
727 {-# INLINE getNumRowsM #-}                                                     [.hs
728 getNumRowsM :: Matrix a -> Int
729 getNumRowsM (M m) = fst $ snd $ bounds m
```

```
730 -- |Gibt zu einer Matrix die Anzahl der Spalten zurück                       GalFld/
731 {-# INLINE getNumColsM #-}                                                     [.hs
732 getNumColsM :: Matrix a -> Int
733 getNumColsM (M m) = snd $ snd $ bounds m
```

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

```
734 {-# INLINE isQuadraticM #-}                                                   GalFld/
735 isQuadraticM :: Matrix a -> Bool                                              [.hs
736 isQuadraticM (Mdiag a) = True
737 isQuadraticM (M m) = uncurry (≡) $ snd $ bounds m
```

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

```
738 -- |Gibt zu einer Matrix den Wert an der Position (row,col) zurück           GalFld/
739 {-# INLINE atM #-}                                                            [.hs
740 atM :: Matrix a -> Int -> Int -> a
741 atM (M m) row col = m!(row,col)
```

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

```

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

747 -- |Gibt zu einer Matrix die i-te Spalte zurück                                     GalFld/
748 {-# INLINE getColM #-}                                                            [.hs]
749 getColM :: Matrix a → Int → [a]
750 getColM (M m) i = [m!(j,i) | j ← [1..k]]
751     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.

Untermatrizen Untermatrizen erhält man wie folgt.

```

752 {-# INLINE subM #-}                                                              GalFld/
753 -- |Gibt zu einer Matrix eine Untermatrix zurück                                  [.hs]
754 -- Input:
755 --     (k0,l0) : erste übernommene Spalte und Zeile
756 --     (k1,l1) : letzte übernommene Spalte und Zeile
757 --     m       : Eingabematrix
758 subM :: Num a ⇒ (Int,Int) → (Int,Int) → Matrix a → Matrix a
759 subM (k0,l0) (k1,l1) (Mdiag x) = subM (k0,l0) (k1,l1) $ genDiagM x $ max k1 l1
760 subM (k0,l0) (k1,l1) (M m)     = M $ subArr (k0,l0) (k1,l1) m

761 {-# INLINE subArr #-}                                                            GalFld/
762 -- |Gibt zu einer Matrix eine Untermatrix zurück                                  [.hs]
763 subArr :: Num a ⇒ (Int,Int) → (Int,Int) → Array (Int, Int) a
764                                           → Array (Int, Int) a
765 subArr (k0,l0) (k1,l1) m = array ((1,1),(k,l))
766     [ ((i-k0+1,j-l0+1) , m!(i,j)) | i ← [k0..k1] , j ← [l0..l1]]
767     where !(k,l) = (k1-k0+1,l1-l0+1)

```

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

```

768 {-# INLINE swapRowsM #-}                                                         GalFld/
769 -- |Vertauscht zwei Zeilen in einer Matrix                                         [.hs]
770 swapRowsM :: Num a ⇒ Int → Int → Matrix a → Matrix a
771 swapRowsM _ _ (Mdiag x) =
772     error "GalFld.Core.Matrix.swapRowsM: Not enough information given"
773 swapRowsM k0 k1 (M m)    = M $ swapRowsArr k0 k1 m

[ ]                                                                                GalFld/
[ ]                                                                                [.hs]

```

2 Implementierung

```
774 {-# INLINE swapColsM #-}
775 -- |Vertauscht zwei Spalten in einer Matrix
776 swapColsM :: Num a => Int -> Int -> Matrix a -> Matrix a
777 swapColsM _ _ (Mdiag x) =
778     error "GalFld.Core.Matrix.swapColsM: Not enough information given"
779 swapColsM 10 11 (M m) = M $ swapColsArr 10 11 m

780 {-# INLINE swapRowsArr #-}
781 -- |Vertauscht zwei Zeilen in einem Array, das zu einer Matrix gehört
782 swapRowsArr :: Num a => Int -> Int -> Array (Int, Int) a -> Array (Int, Int) a
783 swapRowsArr k0 k1 m = array ((1,1),(k,1))
784     [ ((swp i,j) , m!(i,j)) | i <- [1..k] , j <- [1..1]]
785     where (k,1) = snd $ bounds m
786           swp i | i == k0     = k1
787                 | i == k1     = k0
788                 | otherwise = i

789 {-# INLINE swapColsArr #-}
790 -- |Vertauscht zwei Spalten in einem Array, das zu einer Matrix gehört
791 swapColsArr :: Num a => Int -> Int -> Array (Int, Int) a -> Array (Int, Int) a
792 swapColsArr 10 11 m = array ((1,1),(k,1))
793     [ ((i,swp j) , m!(i,j)) | i <- [1..k] , j <- [1..1]]
794     where (k,1) = snd $ bounds m
795           swp j | j == 10     = 11
796                 | j == 11     = 10
797                 | otherwise = j
```

GalFld/
[.hs

GalFld/
[.hs

2.4.2 Zweiwertige Operationen auf Matrizen

Addition Die Addition zweier Matrizen erklärt sich von selbst.

```
798 {-# INLINE addM #-}
799 addM :: (Num a) => Matrix a -> Matrix a -> Matrix a
800 addM (Mdiag x) (Mdiag y) = Mdiag (x+y)
801 addM (Mdiag x) m         = addM m (genDiagM x (getNumRowsM m))
802 addM m (Mdiag y)         = addM m (genDiagM y (getNumRowsM m))
803 addM (M x) (M y)         | boundTest = M $ array (bounds x)
804     [(idx,x!idx + y!idx) | idx <- indices x]
805     | otherwise =
806     error "GalFld.Core.Matrix.addM: not the same Dimensions"
807     where boundTest = bounds x == bounds y
```

GalFld/
[.hs

Multiplikation Die Multiplikation wurde nach dem Standardverfahren implementiert.

```
808 {-# INLINE multM #-}
809 multM :: (Num a) => Matrix a -> Matrix a -> Matrix a
810 multM (Mdiag x) (Mdiag y) = Mdiag (x*y)
811 multM (Mdiag x) m         = multM (genDiagM x (getNumRowsM m)) m
812 multM m (Mdiag x)         = multM m (genDiagM x (getNumColsM m))
813 multM (M m) (M n)         | k' == 1 = M $ array ((1,1),(k,1'))
```

GalFld/
[.hs

```

814  [(i,j), sum [m!(i,k) * n!(k,j) | k <- [1..l]]] | i <- [1..k] , j <- [1..l']]
815      | otherwise =
816  error "GalFld.Core.Matrix.multM: not the same Dimensions"
817  where ((_,_), (k,l)) = bounds m
818      ((_,_), (k',l')) = bounds n

```

2.4.3 Lineare Algebra

Transponieren

```

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

```

GalFld/
[.hs]

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

```

825 -- |Berechnet die Zeilenstufenform einer Matrix
826 {-# INLINE echelonM #-}
827 echelonM :: (Show a, Eq a, Num a, Fractional a) ⇒ Matrix a → Matrix a
828 echelonM (Mdiag n) = Mdiag n
829 echelonM (M m)     = M $ echelonM' m
830 where echelonM' :: (Show a, Eq a, Num a, Fractional a) ⇒
831     Array (Int,Int) a → Array (Int,Int) a
832     echelonM' m | k ≡ 1      = arrElim m
833                 | l ≡ 1      = arrElim m
834                 | hasPivot   = echelonM' $ swapRowsArr 1 (minimum lst) m
835                 | noPivot    = echelonM'_noPivot m
836                 | otherwise  = echelonM'_Pivot m
837     where !(k,l) = snd $ bounds m
838           !lst   = [i | i <- [1..k], m!(i,1) ≠ 0]
839           !hasPivot = m!(1,1) ≡ 0 && not (null lst)
840           !noPivot  = m!(1,1) ≡ 0 && null lst

```

GalFld/
[.hs]

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`.

```

841 {-# INLINE arrElim #-}
842 -- |Zieht die erste Zeile passend von allen anderen ab, eliminiert also in
843 -- jeder außer der ersten Zeile den ersten Eintrag der Zeile
844 arrElim :: (Eq a, Num a, Fractional a) ⇒ Array (Int, Int) a
845                                             → Array (Int, Int) a
846 arrElim m | m!(1,1) ≡ 0 = m
847             | otherwise =
848     (m // [ ((1,j),m!(1,j)/m!(1,1)) | j <- [1..l]])
849     // [ ((i,j), m!(i,j) - m!(i,1) / m!(1,1) * m!(1,j)) | j <- [1..l],

```

GalFld/
[.hs]


```

850         i ← [2..k]]
851     where !(k,l) = snd $ bounds m

```

Beispiel 2.14 Wir versuchen einmal die „Telefonmatrix“

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

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

```

echelonM' (k,l)=(3,3) m=
25 35 45
45 05 15
15 25 35
      →(1,1)≠0
echeonM'_Pivot m'=
15 45 25
05 45 35
05 35 15
echelonM' (k,l)=(2,2) m=
45 35
35 15
      →(1,1)≠0
echeonM'_Pivot m'=
15 25
05 05

```

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

```

15 45 25
05 15 25
05 05 05

```

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, 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. [15, Abschnitt Basis]). Dies lässt sich durch Transponieren natürlich leicht auf die Berechnung einer Zeilenstufenform zurückführen.

```

871 -- |Berechnet den Kern einer Matrix, d.h.                                     GalFld/
872 -- kernelM gibt eine Matrix zurück, deren Spalten eine Basis des          [.hs]
873 -- des Kerns sind
874 {-# INLINE kernelM #-}
875 kernelM :: (Show a, Eq a, Num a, Fractional a) => Matrix a -> Matrix a
876 kernelM (Mdiag m) = error "GalFld.Core.Matrix.kernelM: No kernel here"
877 kernelM m         = M $ array ((1,1), (k,lzs))
878                        [ ((i,j),b!(i,zs!!(j-1))) | i ← [1..k], j ← [1..lzs]]
879     where !(k,l) = snd $ bounds $ unM m
880             !mfull = transposeM $ echelonM $

```

```

881         transposeM $ m <=> genDiagM 1 k
882     !a      = subArr (1,1) (k,1) $ unM mfull
883     !b      = subArr (k+1,1) (k+k,1) $ unM mfull
884     !zs     = [j | j <- [1..1], and [a!(i',j) == 0 | i' <- [j..k]]]
885     !lzs    = length zs

```

Determinante Offensichtlich kann man mit der Zeilenstufenform auch die Determinante einer Matrix berechnen.

```

886 {-# INLINE detM #-}                                     GalFld/
887 -- |Berechne die Determinante effektiver als detLapM aber braucht Fractional [.hs]
888 detM :: (Eq a, Num a, Fractional a) => Matrix a -> a
889 detM (Mdiag 0) = 0
890 detM (Mdiag 1) = 1
891 detM (Mdiag _) =
892     error "GalFld.Core.Matrix.detM: Not enough information given"
893 detM m          | isQuadraticM m = detArr $ unM m
894                  | otherwise      =
895     error "GalFld.Core.Matrix.detM: Matrix not quadratic"
896     where {-# INLINE detArr #-}
897           -- |detM auf Array ebene
898           detArr :: (Eq a, Num a, Fractional a) => Array (Int, Int) a -> a
899           detArr m | k == 1          = m!(1,1)
900                     | m!(1,1) == 0  = - detArrPivot m
901                     | otherwise     = (m!(1,1) *) $ detArr $ subArr (2,2) (k,1) $
902                                     arrElim m
903           where !(k,1) = 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.⁴

```

904 {-# INLINE detLapM #-}                                   GalFld/
905 -- |Berechne die Determinante ohne Nutzen von Fractional a [.hs]
906 detLapM :: (Eq a, Num a) => Matrix a -> a
907 detLapM (Mdiag 0) = 0
908 detLapM (Mdiag 1) = 1
909 detLapM (Mdiag _) =
910     error "GalFld.Core.Matrix.detLapM: Not enough information given"
911 detLapM m | isQuadraticM m = detLapM' $ unM m
912           | otherwise      = 0
913 {-# INLINE detLapM' #-}
914 detLapM' :: (Eq a, Num a) => Array (Int, Int) a -> a

```

⁴Man hätte auch die Berechnung der Smithschen-Normalform implementieren können, die ebenfalls ohne **Fractional** ausgekommen wäre. Jedoch haben wir uns entschieden darauf zu verzichten, da im vorliegenden Anwendungsfall der endlichen Körper stets Inverse zur Verfügung stehen.

```

915 detLapM' m | b ≡ (1,1) = m!(1,1)
916 | otherwise =
917   sum [(-1)^(i-1) * m!(i,1) * detLapM' (getSubArr i) | i ← [1..fst b]]
918   where !b = snd $ bounds m
919   {-# INLINE getSubArr #-}
920   getSubArr i = array ((1,1),(fst b-1,snd b-1)) $
921     [((i',j'),m!(i',j'+1)) | i' ← [1..(i-1)]
922       , j' ← [1..(snd b - 1)]]
923     # [((i',j'),m!(i'+1,j'+1)) | i' ← [i..fst b - 1]]
924       , j' ← [1..(snd b - 1)]]

```

2.4.4 Weiteres

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

```

923 {-# INLINE getAllM #-}
926 -- |Gibt eine Liste aller Matrizen, welche Einträge aus einer Liste besitzen
927 -- und eine gewisse gröÙe haben, zurrück
928 getAllM :: [a] → (Int,Int) → [Matrix a]
929 getAllM cs (k,l) = map fromListsM $ rowMs k
930   where lines = lines' 1
931         lines' n | n ≡ 1 = [[y] | y ← cs]
932                 | otherwise = [y:ys | y ← cs, ys ← lines' (n-1) ]
933         rowMs n | n ≡ 1 = [[y] | y ← lines]
934                 | otherwise = [y:ys | y ← lines, ys ← rowMs (n-1) ]

```

GalFld/
[.hs]

2.5 Faktorisierung von Polynomen über endlichen Körpern

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

Faktorisierungsalgorithmen Grundsätzlich sind alle Algorithmen, die in Kapitel 3 beschrieben und der konkreten Faktorisierung von Polynomen dienen, als `Polynom a → [(Int, Polynom a)]` 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/
[.hs]

```

936 obviousFactor :: (Show a, Num a, Eq a) => Polynom a -> [(Int,Polynom a)]
937 obviousFactor f | isNullP f      = [(1,nullP)]
938                 | uDegP f ≤ 1    = [(1,f)]
939                 | hasNoKonst fs = factorX
940                 | otherwise     = toFact f
941 where fs = p2Tup f
942       -- Teste, ob ein konstanter Term vorhanden ist
943       hasNoKonst ms | fst (last ms) == 0 = False
944                   | otherwise           = True
945       -- Hier kann man d mal X ausklammern
946       factorX | g == 1 = [(d,pTupUnsave [(1,1)])]
947              | otherwise = [(d,pTupUnsave [(1,1)]), (1,g)]
948       where d = fst $ last fs
949             g = pTupUnsave $ map (A.first (\x -> x-d)) fs

```

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

```

950 import Debug.Trace                                     GalF1d/
951 -----                                                [.hs]
952 -- |Erzeugt eine triviale Faktoriesierung zu einem Polynom
953 toFact :: Polynom a -> [(Int,Polynom a)]
954 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. quadratfrei 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. Dazu gibt es den nachstehen Wrapper `appFact`.

```

955 -- |Nimmt eine Faktoriesierung und wendet auf diese einen gegebenen GalF1d/
956 -- Faktoriesierungsalgorithmus an                               [.hs]
957 appFact :: (Eq a, Num a) =>
958   (Polynom a -> [(Int,Polynom a)]) -> [(Int,Polynom a)] -> [(Int,Polynom a)]
959 appFact alg = withStrategy (parList rpar) . concatMap
960   (uncurry appFact')
961 where appFact' i f | isNullP f      = [(i,nullP)]
962                  | uDegP f ≤ 1    = [(i,f)]
963                  | otherwise     = potFact i (alg f)

```

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

```

964 -- |Ersetzt eine Faktoriesierung, durch die n-te Potenz dieser Faktoriesierung GalF1d/
965 potFact :: (Num a) => Int -> [(Int,Polynom a)] -> [(Int,Polynom a)]       [.hs]
966 potFact _ []      = []
967 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 `appA` wobei A für den jeweiligen konkreten Algorithmus steht.

```

968 appObFact :: (Show a, Num a, Eq a) => [(Int,Polynom a)] -> [(Int,Polynom a)] GalFld/
969 appObFact = appFact obviousFactor [.hs

971 appSff :: (Show a, FiniteField a, Num a, Fractional a) => GalFld/
972 [(Int,Polynom a)] -> [(Int,Polynom a)] [.hs
973 appSff = appFact sff

974 appBerlekamp :: (Show a, FiniteField a, Num a, Fractional a) => GalFld/
975 [(Int,Polynom a)] -> [(Int,Polynom a)] [.hs
976 appBerlekamp = appFact berlekampFactor

977 appBerlekamp2 :: (Show a, FiniteField a, Num a, Fractional a) => GalFld/
978 [(Int,Polynom a)] -> [(Int,Polynom a)] [.hs
979 appBerlekamp2 = appFact berlekampFactor2

```

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 `appA` verschiedene Tupel entstehen, deren eigentlicher Faktor jedoch der gleiche ist. Da diese erstmal nicht erkannt werden, gibt es `aggFact`, um am Ende in einer Faktorisierung nur paarweise verschiedene Faktoren zu haben.

```

980 -- | Fasst in einer Faktorisierung gleiche Terme zusammen GalFld/
981 aggFact :: (Num a, Eq a) => [(Int,Polynom a)] -> [(Int,Polynom a)] [.hs
982 aggFact l = [(sum [i | (i,g) <- l, f≡g],f) | f <- nub [f | (_,f) <- l],
983 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.

Es sind auch noch Funktionen implementiert, die

- ein Datentyp der Klasse `ShowTex` zu einem PNG rendern können bzw.
- einen String mit \LaTeX code rendern.

- Sowie eine Funktion, die das Programm `sxiv` nutzt, um diese PNG anzuzeigen.

Diese drei Funktionen sind leider nur unter Linux verfügbar.

```

984 -- |wie renderRawTex, nur dass zunächst ShowTex aufgerufen wird.
985 renderTex :: (ShowTex a) => a -> IO ()
986 renderTex = renderRawTex . showTex
987 -----
988 -- |Nutze sxiv um das erzeugte Bild anzuzeigen
989 viewRendered = do createProcess (shell ("sxiv " # outputPNG))
990                 return ()
991 #else
992 outputPNG = undefined
993 renderTex = undefined
994 renderRawTex = undefined
995 viewRendered = undefined
996 #endif

```

GalFld/
[.hs]

```

998 -- |Nimmt einen Latex-String und packt diesen in ein minimales Latex-Dokument,
999 -- rendert dieses und wandelt es danach in ein Bild um, wobei unnötiger Rand
1000 -- entfernt wird
1001 renderRawTex :: String -> IO ()
1002 renderRawTex x = do createProcess (shell cmd)
1003                  return ()
1004      where cmd = "latex -halt-on-error -output-directory " # outputDIR # " "
1005                + "'\\documentclass[12pt]{article}"
1006                + "\\pagestyle{empty}" # "\\usepackage{amsmath}"
1007                + "\\begin{document}"
1008                + "\\begin{multline*}" # x # "\\end{multline*}"
1009                + "\\end{document}" > /dev/null ; "
1010                + "dvipng -gamma 2 -z 9 -T tight -bg White " -- -bg Transparent
1011                + "-o " # outputPNG # " " # outputDVI # " > /dev/null"
1012 -----
1013 -- |Nutze sxiv um das erzeugte Bild anzuzeigen
1014 viewRendered = do createProcess (shell ("sxiv " # outputPNG))
1015                 return ()
1016 #else
1017 outputPNG = undefined
1018 renderTex = undefined
1019 renderRawTex = undefined
1020 viewRendered = undefined
1021 #endif

```

GalFld/
[.hs]

2.6.2 Spezielle Polynome und zahlentheoretische Funktionen

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

Vorgreifend auf ?? 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 2.15 (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

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

die Menge der d -ten primitiven Einheitswurzeln in \mathbb{F}_q ist.⁵

Definition 2.16 (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 2.17 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.

```

1023 -- |Primfaktorzerlegung (enthält Vielfache!)
1024 -- aus http://www.haskell.org/haskellwiki/99_questions/Solutions/35
1025 primFactors :: Int -> [Int]
1026 primFactors 1 = []
1027 primFactors n = let divisors = dropWhile ((\x -> x == 0) . mod n)
1028                  [2 .. ceiling $ sqrt $ fromIntegral n]
1029                  in let prime = if null divisors then n else head divisors
1030                  in (prime :) $ primFactors $ div n prime

1031 isPrime :: Int -> Bool
1032 isPrime n = 1 == length (primFactors n)

```

⁵Es sei vorausgesetzt, dass dem Leser nicht explizit definierten Termini bekannt sind.

```

1033 -- |Teiler von n
1034 divisors :: Int → [Int]
1035 divisors n | n == 1      = div'
1036             | otherwise = div' * [n]
1037     where div' = 1 : filter ((/=0) . rem n) [2 .. n `div` 2]

1039 -- |Möbius-Funktion µ mit
1040 --   µ(n) = (-1)^k, falls n quadratfrei, k = #Primfaktoren, 0 sonst
1041 moebFkt :: Int → Int
1042 moebFkt n | facs == nub facs && even (length facs) = 1
1043           | facs == nub facs && odd  (length facs) = -1
1044           | otherwise                             = 0
1045     where facs = primFactors n

1046 -- |Die Kreisteilungspolynome
1047 --   gibt das n-te Kreisteilungspolynom über dem Körper dem e zu Grunde liegt
1048 cyclotomicPoly :: (Show a, Fractional a, Num a, FiniteField a) ⇒
1049                 Int → a → Polynom a
1050 cyclotomicPoly 1 e = pTupUnsave [(1,1),(0,-1)]
1051 cyclotomicPoly n e
1052   | isPrime n = pTupUnsave $ map (λi → (i,1)) $ reverse [0..n-1]
1053   | otherwise = foldl (@/) numerator $ map fst $ filter (λ(_,m) → m==(-1)) 1
1054   where numerator = product $ map fst $ filter (λ(_,m) → m==1) 1
1055         1 = [(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 2.18 Wie man sich leicht überlegt, sind alle primitiven Elemente eines Körpers \mathbb{F}_q , also $u \in \mathbb{F}_q^*$ mit $\text{ord } u = q - 1$, 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 $\text{Ord}_q(u)$ gleich $X^n - 1$ ist, als Nullstellen der Pi-Polynome schreiben.

Definition 2.19 (*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 2.20 (Pi-Polynom, [6, Definition 3.4]) Sei $f \in \mathbb{F}_q[X]$ ein monischer Teiler von $X^n - 1$. Notiere

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

Dann heit

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

das *Pi-Polynom* zu f ber \mathbb{F}_q .

Definition 2.21 Fr $f, g \in \mathbb{F}_q[X]$ 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 vollstndige 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

$$P_{f_1 \dots f_i} := (P_{f_1 \dots f_{i-1}} \odot f_i) P_{f_1 \dots f_{i-1}}^{-1}$$

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 bertragen.

```

1065 -- | Gibt das Pi-Polynom zu f
1066 -- f muss ein monischer Teiler von x^m -1 ber F_q sein
1067 piPoly :: (Show a, Num a, Fractional a, FiniteField a) =>
1068                                     Polynom a -> Polynom a
1069 piPoly f
1070   | isSqfree  = piSqFree
1071   | otherwise = piSqFree `odot` fNonSqFree
1072 where -- P_(tau f), wobei tau f der quadratfreie Teil von f ist
1073       piSqFree = foldl (\p f -> (p `odot` f) @/ p) pFst (map snd $ tail facs)
1074       -- Faktorisierung von f
1075       facs = factorP f
1076       -- Start der Rekursion mit P_(f1)
1077       pFst = assozPoly (snd $ head facs) @/ pTupUnsave [(1,1)]
1078       -- Definition von odot
1079       odot f g = evalPInP f $ assozPoly g
1080       -- Test auf quadratfrei
1081       isSqfree = all (≡1) $ map fst facs
1082       -- f / tau(f)
1083       fNonSqFree = unFact $ map (A.first (\i -> i-1)) facs

```

GalFld/
[.hs]

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 $i = 1, \dots, m$ gilt, dass $r_i(X)$ monisch und quadratfrei ist und für alle $i, j = 1, \dots, m$, $i \neq j$, 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 $\text{ggT}(f(X), f'(X)) \neq 0$, wobei $f'(X)$ die formale Ableitung von f meint. Damit kann man folgern, dass $\text{ggT}(f(X), f'(X)) = 1$ 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 [cohen:algebra] 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 $\text{char } \mathbb{F} = p$ für eine Primzahl p und der Frobenius $\sigma : \mathbb{F} \rightarrow \mathbb{F}, x \mapsto x^p$ 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 $f(X) = a_n X^n + \dots + a_0$, 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

$$f(X) = a_{pk} X^{pk} + \dots + a_p X^p + b_0.$$

Definiere nun

$$b(X) = b_k X^k + \dots + b_1 X + b_0 \quad \text{mit} \quad b_i = a_{pi}^{\frac{1}{p}} \quad i = 0, \dots, k.$$

Da wir wissen, dass der Frobenius $\mathbb{F} \rightarrow \mathbb{F}, x \mapsto x^p$ ein Automorphismus auf \mathbb{F} ist, ist $(\cdot)^{\frac{1}{p}}$ 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)
 $c(X) := \text{ggT}(f(X), b(X))$
 $w(X) := f(X)/c(X)$
 while $w(X) \neq 1$ **do** (1.1)
 $y(X) := \text{ggT}(w(X), c(X))$, $z(X) := w(X)/y(X)$
 $r(X) := r(X) \cdot z(X)^i$, $i := i + 1$
 $w(X) := y(X)$, $c(X) := c(X)/y(X)$
 endwhile
 if $c(X) \neq 1$ **then** (1.2)
 $c(X) := c(X)^{\frac{1}{p}}$
 $r(X) := r(X) \cdot (\text{SFF}(c(X)))^p$
 endif
else (2)
 $f(X) := f(X)^{\frac{1}{p}}$
 $r(X) := (\text{SFF}(f(X)))^p$
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

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

und wir können folgern,

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

wobei $A = \{i = 1, \dots, m : p \nmid i\}$. 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 $A = \{i_1, \dots, i_k\}$ in aufsteigend sortierter Reihenfolge haben wir

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

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

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 auch offensichtlich, betrachtet man erneut Proposition 3.5 und die Umsetzung in Schritt 1.2. \square

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. [13, Theorem 6.1 mit Corollary 4.5]. \square

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 die nachstehender Satz, die 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

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

und die ggTs sind paarweise teilerfremd.

Beweis. [13, 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 offenbar 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:

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

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

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

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 degenierte, 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 $\text{ggT}(f(X), g(X) - a) \neq f(X)$ 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*

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

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 \mathcal{B}_f Die Basis des Berlekamp-raumes berechnen wir mit den in Abschnitt 2.4 vorgestellten Methoden der linearen Algebra.

```

1104 -- |Berechnet eine Basis des Berlekamp-raums zu f,
1105 -- d.h. gibt eine Matrix zurück, deren Zeilen gerade den Berlekamp-raum
1106 -- aufspannen bzgl der kanonischen Basis { 1, x, x^2, x^3, ... }
1107 berlekampBasis :: (Show a, Fractional a, Num a, FiniteField a)
1108                  ⇒ Polynom a → Matrix a
1109 berlekampBasis f = --trace ("mods = \n"++show (fromListsM [red i | i <- [0..(n-1)]]) $
1110   transposeM $ kernelM $ transposeM $!
1111   fromListsM [red i | i <- [0..(n-1)]] - genDiagM 1 n
1112   where !n      = fromJust $ degP f
1113         !q      = elemCount a
1114         !a      = getReprP f
1115         {-# INLINE red #-}
1116         red i = takeFill 0 n $ p2List $ modByP (pTupUnsave [(i*q,1)]) f

```

GalFld/
[.hs]

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, was aber der Funktionalität von Haskell widerspricht.

```

1117 -- |Faktoriert ein Polynom f über einem endlichen Körper
1118 -- Voraussetzungen: f ist quadratfrei
1119 -- Ausgabe: Liste von irreduziblen, pw teilerfremden Polynomen
1120 berlekampFactor :: (Show a, Fractional a, Num a, FiniteField a)
1121                  ⇒ Polynom a → [(Int,Polynom a)]
1122 berlekampFactor f | isNullP f      = []
1123                   | uDegP f < 2    = [(1,f)]
1124                   | otherwise      = berlekampFactor' f m
1125   where !m = berlekampBasis f
1126         {-# INLINE berlekampFactor' #-}
1127         berlekampFactor' :: (Show a, Num a, Fractional a, FiniteField a)
1128                           ⇒ Polynom a → Matrix a → [(Int,Polynom a)]
1129         berlekampFactor' f m | uDegP f ≤ 1      = [(1,f)]
1130                              | getNumRowsM m ≡ 1 = [(1,f)]
1131                              | otherwise          =
1132                                concat [berlekampFactor' g (newKer m g) | g <- gs]
1133   where {-# INLINE gs #-}
1134         gs = [x | x <- [ggTP f (h - pKonst s)
1135                        | s <- elems (getReprP f)] , x ≠ 1]
1136         {-# INLINE h #-}
1137         h = pList $ getRowM m 2
1138         {-# INLINE newKer #-}
1139         newKer m g = fromListsM $! take r m'
1140         where !(k,l) = boundsM m
1141               !m'    = toListsM $ echelonM $ fromListsM
1142                       [takeFill 0 l $ p2List $
1143                        modByP (pList (getRowM m i)) g | i <- [1..k]]

```

GalFld/
[.hs]


```

1144      !r      = k-1- fromMaybe (-1) (findIndex (all (≡0))
1145                                          $ reverse m')

```

Die Berechnung der neuen Basis bei der rekursive 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

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

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

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

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

$$\begin{array}{c|cccc}
 a \in \mathbb{F}_q & 0 & 1 & 2 & 3 & 4 \\
 \hline
 \text{ggT}(f(X), h(X) - a) & X^2 + 2 & X^2 + 4X + 3 & 1 & 1 & X + 2
 \end{array}$$

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 Berlekamp-raumes 1 ist. Für f_2 wählen wir $h(X) = X + 2$ und erhalten

$$\frac{a \in \mathbb{F}_q \mid \begin{array}{c|cccc} & 0 & 1 & 2 & 3 & 4 \\ \hline \text{ggT}(f_2(X), h(X) - a) & 1 & 1 & X + 3 & 1 & X + 1 \end{array}}{.}$$

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

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

3.2.4 Alternative Implementierungen

Es ist offensichtlich, dass man verschiedene Wahlen hat, den rekursiven Aufruf des Berlekamp-Algorithmus zu gestalten. Die zweite Möglichkeit, die wir implementiert haben und hier aufzeigen möchten, besteht darin, lediglich auf den ersten nicht-trivialen ggT zu warten und in zwei rekursiven Aufrufen mit eben jenem ggT und seinem Kofaktor in $f(X)$ zu enden.

```

1146 -- |Faktorisiert ein Polynom f über einem endlichen Körper
1147 -- Voraussetzungen: f ist quadratfrei
1148 -- Ausgabe: Liste von irreduziblen, pw teilerfremden Polynomen
1149 berlekampFactor2 :: (Show a, Fractional a, Num a, FiniteField a)
1150                  => Polynom a -> [(Int, Polynom a)]
1151 berlekampFactor2 f | isNullP f      = []
1152                   | uDegP f < 2     = [(1,f)]
1153                   | otherwise      = berlekampFactor' f m
1154 where !m = berlekampBasis f
1155       {-# INLINE berlekampFactor' #-}
1156       berlekampFactor' :: (Show a, Num a, Fractional a, FiniteField a)
1157                        => Polynom a -> Matrix a -> [(Int, Polynom a)]
1158       berlekampFactor' f m | uDegP f <= 1      = [(1,f)]
1159                           | getNumRowsM m == 1 = [(1,f)]
1160                           | otherwise          =
1161                             berlekampFactor' g n * berlekampFactor' g' n'
1162       where {-# INLINE g #-}
1163             g = head [x | x <- [ggTP f (h - pKonst s)
1164                               | s <- elems (getReprP f)] , x /= 1]
1165             {-# INLINE g' #-}
1166             g' = f @/ g
1167             {-# INLINE h #-}
1168             h = pList $ getRowM m 2
1169             {-# INLINE n #-}
1170             n = newKer m g
1171             {-# INLINE n' #-}
1172             n' = newKer m g'
1173             {-# INLINE newKer #-}
1174             newKer m g = fromListsM $! take r m'
1175             where !(k,l) = boundsM m
1176                   !m'    = toListsM $ echelonM $ fromListsM
1177                             [takeFill 0 l $ p2List $
1178                               modByP (pList (getRowM m i)) g | i <- [1..k]]
1179                   !r      = k-1- fromMaybe (-1) (findIndex (all (==0))
1180                                                             $ reverse m')

```

Ein kleiner Vergleich

Es hat sich herausgestellt, dass beide Varianten nahezu identische Laufzeiten haben. Bei einem kleinen Vergleich von je 30 Polynomen eines Grades stellt sich heraus, dass letztere Variante in kleinen Graden schneller ist, wohingegen die erste Variante in höheren Graden effizienter arbeitet. Abbildung 3.1 fasst die Ergebnisse zusammen.

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 [12] 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. $\text{ggT}(g(X), X^{q^{n_i}} - X) = 1$ für alle $i = 1, \dots, k$.

Beweis. [12, Lemma 1]. Dort zwar nur für \mathbb{Z}_p , jedoch lässt sich der Beweis für beliebiges \mathbb{F}_q problemlos erweitern. \square

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

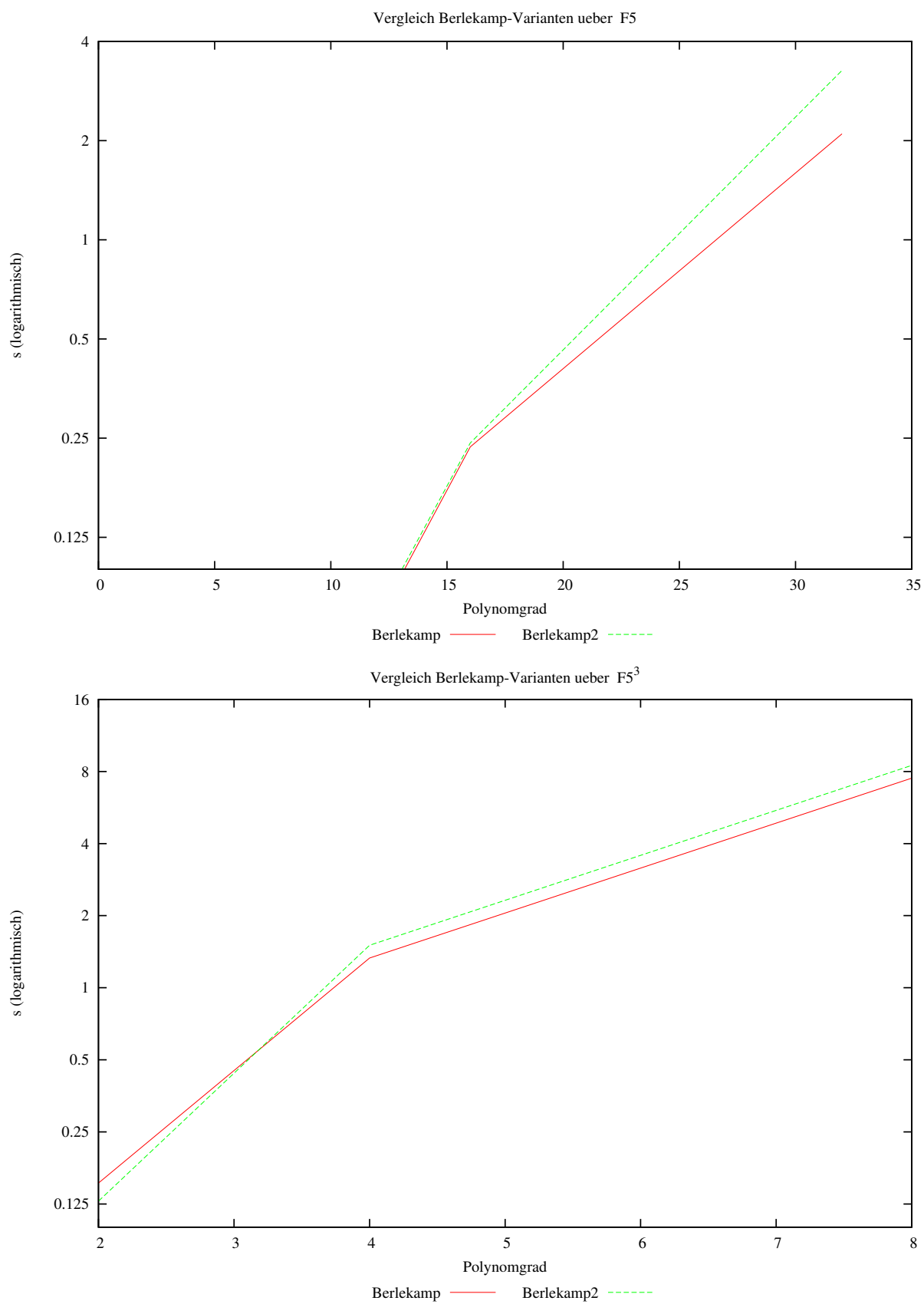
```

1181 -- |Rabin's Irreduzibilitätstest
1182 --   Ausgabe: True, falls f irreduzibel, False, falls f reduzibel
1183 --
1184 --
1185 --   Algorithm Rabin Irreducibility Test
1186 --   Input: A monic polynomial f in Fq[x] of degree n,
1187 --          p1, ..., pk all distinct prime divisors of n.
1188 --   Output: Either "f is irreducible" or "f is reducible".
1189 --   Begin
1190 --       for j = 1 to k do
1191 --           n_j := n / p_j;
1192 --       for i = 1 to k do
1193 --           h := x^(q^n_i) - x mod f;
1194 --           g := gcd(f, h);
1195 --           if g /= 1, then return 'f is reducible' and STOP;
1196 --       end for;
1197 --       g := x^(q^n) - x mod f;
1198 --       if g = 0, then return "f is irreducible",

```

GalFld/
[.hs

Abbildung 3.1: Vergleich der Berlekamp-Varianten über \mathbb{F}_5 und \mathbb{F}_{5^3}



3 Algorithmen auf Polynomen über endlichen Körpern

```

1199 --           else return "f is reducible"
1200 --     end.
1201 --   vgl http://en.wikipedia.org/wiki/Factorization_of_polynomials_over_
1202 --       finite_fields#Irreducible_polynomials
1203 rabin :: (Show a, FiniteField a, Num a, Fractional a, Eq a) => Polynom a -> Bool
1204 rabin f | isNullP f = False
1205         | otherwise = rabin' f ns
1206   where ns = map (\p -> d `quot` p) $ nub $ factor d
1207         d   = uDegP f
1208         q   = elemCount $ getReprP f
1209         pX  = pTupUnsave [(1,1)]
1210         -- eigentlicher Rabin für den letzten Test mit  $x^{(q^n)} - x$ 
1211         rabin' f [] = isNullP g
1212         where g = (h' - pX) `modByP` f
1213               h' = modMonom q d f
1214         -- eigentlicher Rabin für  $x^{(q^{n_j})} - x$  mit  $n_j = n / p_j$ 
1215         rabin' f (n:ns) | g /= pKonst 1 = False
1216                       | otherwise = rabin' f ns
1217         where g = ggTP f (h' - pX)
1218               h' = modMonom q n f

```

Zu beachten gilt, dass die Reduktion $\text{mod}f(X)$ 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 $X^{q^{n_i}} \text{mod} f(X)$ berechnet.

```

1219 -- | Schnelles Modulo für Monome, d.h. berechnet
1220 --    $x^{(q^d)} \text{mod} f$ 
1221 modMonom :: (Show a, Num a, Eq a, Fractional a) =>
1222           Int -> Int -> Polynom a -> Polynom a
1223 modMonom q d = modMonom' n
1224   where n = toInteger q ^ toInteger d
1225         modMonom' n f
1226           | n < toInteger df
1227             = pTupUnsave [(fromInteger n,1)]
1228           | even n = g `modByP` f
1229           | otherwise = multMonomP 1 g `modByP` f
1230   where df = uDegP f
1231         m = n `quot` 2
1232         g = h*h
1233         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:

```

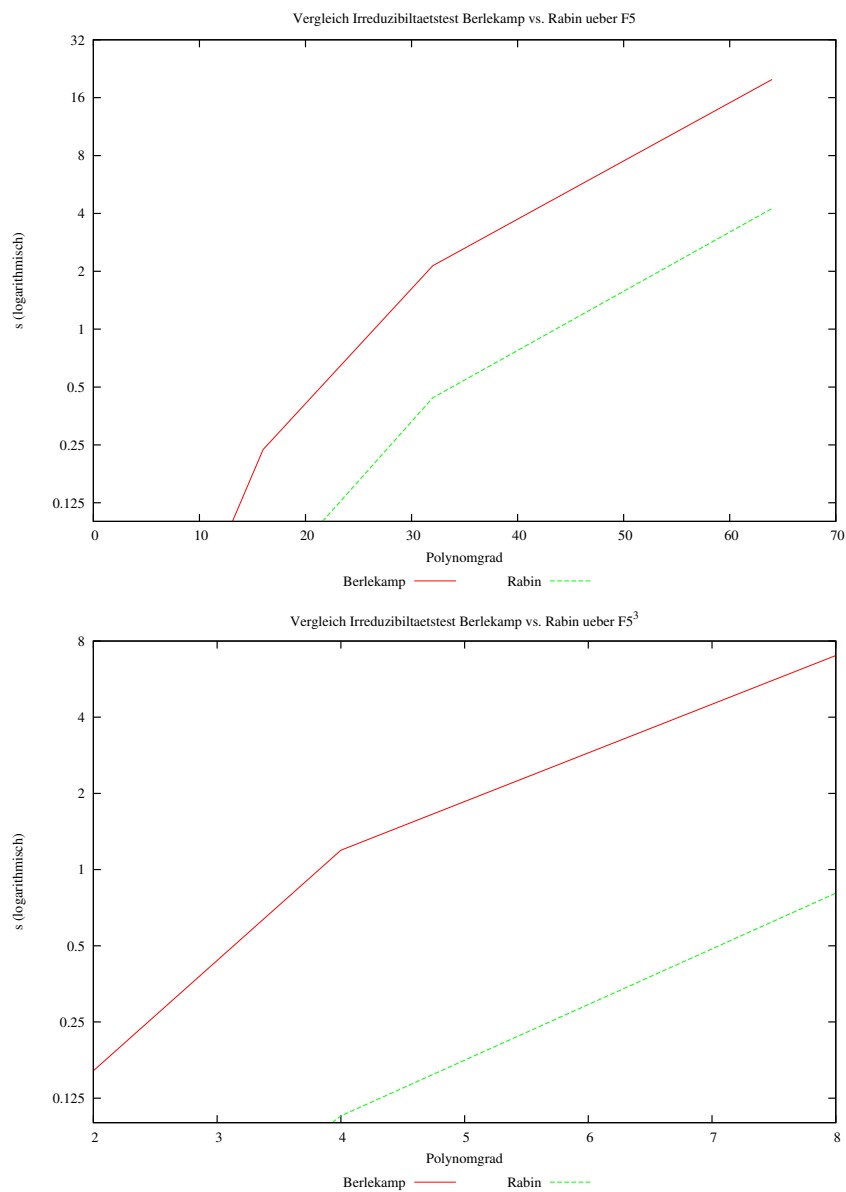
1235 -- | Primfaktorzerlegung
1236 --   aus http://www.haskell.org/haskellwiki/99_questions/Solutions/35
1237 factor :: Int -> [Int]
1238 factor 1 = []
1239 factor n = let divisors = dropWhile ((/= 0) . mod n) [2 .. ceiling $ sqrt $ fromIntegral n]
1240           in let prime = if null divisors then n else head divisors
1241           in (prime :) $ factor $ div n prime

```

3.3.1 Ein kleiner Vergleich

Auch wenn der Vergleich einer vollständigen Faktorisierung via Berlekamp mit Rabin als Irreduzibilitätstest ob des Mehraufwands nicht ganz fair ist, so wollen wir ihn doch anführen. Abbildung 3.2 deutlich, dass die bloße Information der Irreduzibilität viel leichter zu gewinnen ist, als die gesamte Faktorisierung. (Wer hätte das gedacht!)

Abbildung 3.2: Vergleich Berlekamp vs Rabin als Irreduzibilitätstest



4 Beispiel: 1

5 Beispiel: Primitiv-normale-Elemente

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 2.18 erläutert, sind die Nullstellen des Pi-Polynom 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!

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

Imports zum messen der Ausführzeit und zum verarbeiten von Input Parametern.

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

Importiere auch die nötige Bibliothek `GalFld` und `GalFld.More.SpecialPolys` welches standardmäßig nicht enthalten ist, da es eine sehr spezielle Funktion enthält.

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

Wir Erzeugen uns einen Primkörper mit charakteristik `p` mit dem Namen `PF`.

```
$(genPrimeField 2 "PF")  
  
pf = 1::PF  
p = charakteristik pf
```

Wir machen uns eine Datenstruktur `T` in der wir später Information speichern wollen.

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

Hier nun die Funktion `genPrimNorm`, die zu einem gegebenem `Int` als Grad die ganze Arbeit erledigt und die notwendigen Polynome generiert und den ggT dieser faktorisiert.

```
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)

```

Nun definieren wir noch eine praktische Funktion `if'`, die leider in Prelude fehlt.

```

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

```

In der `main` wird alles zusammengefügt und schön formatiert ausgegeben.

```

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_ (λn → do
    st ← getCPUTime
    let gpn = genPrimNorm n
    putInfo $ fst gpn
    putPolys $ snd gpn
    putTime st ) indxs
    where putInfo (T n cP cN cPN) = do
      putStrLn $ "In F" # show p # "^" # show n # " über F" # show p
      # " 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"

```

Literaturverzeichnis

- [1] Mohammad Bavarian. „Lecture 6“. In: Madhu Sudan. *6.S897 Algebra and Computation*. Vorlesungsskript. 2012. URL: <http://people.csail.mit.edu/madhu/ST12/scribe/lect06.pdf>.
- [2] J.S. Cohen. *Computer algebra and symbolic computation: mathematical methods*. Ak Peters Series. AK Peters, 2003. ISBN: 9781568811598.
- [3] K.O. Geddes, S.R. Czapor und G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992. ISBN: 9780585332475. 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] Dirk Hachenberger. „On Primitive and Free Roots in a Finite Field.“ In: *Appl. Algebra Eng. Commun. Comput.* 3 (1992), S. 139–150.
- [7] G. Hutton. *Programming in Haskell*. Cambridge University Press, Jan. 2007.
- [8] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. 1st. San Francisco, CA, USA: No Starch Press, 2011. ISBN: 1593272839, 9781593272838.
- [9] Simon Marlow u. a. *Haskell 2010 Language Report*. http://www.haskell.org/haskellwiki/Language_and_library_specification. 2010.
- [10] Marc Moreno Maza. „From Newton to Hensel“. In: *Foundations of Computer Algebra*. Vorlesungsskript. 2008. URL: <http://www.csd.uwo.ca/~moreno/CS424/Lectures/FastDivisionAndGcd.ps.gz>.
- [11] Simon 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.
- [12] M.O. 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>.
- [13] Z.X. Wan. *Lectures on Finite Fields and Galois Rings*. World Scientific, 2003. ISBN: 9789812385703.
- [14] Wikipedia. *Horner-Schema* — *Wikipedia, Die freie Enzyklopädie*. 2014. URL: <http://de.wikipedia.org/w/index.php?title=Horner-Schema&oldid=130454488>.
- [15] Wikipedia. *Kernel (linear algebra)*. 2014. URL: [http://en.wikipedia.org/wiki/Kernel_\(linear_algebra\)](http://en.wikipedia.org/wiki/Kernel_(linear_algebra)).
- [16] 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/>