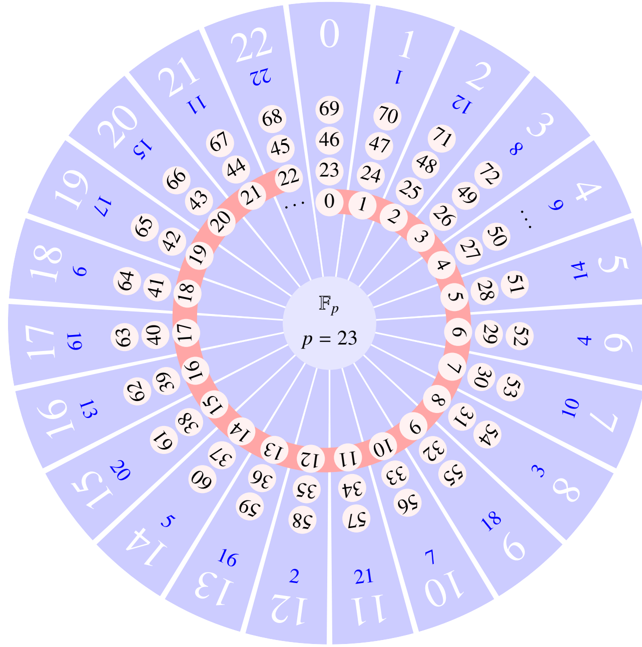
SA
Dokumentation

Arithmetik in endlichen Körpern

Semester: Herbst 2024



Version: 1.0
Datum: 2024-12-20 13:33:56Z

Projekt Team: Ali Al-Kubaisi
Simon Amberg

Projekt Betreuer: Andreas Müller
Thomas Kämpfer

Departement Informatik
OST – Ostschweizer Fachhochschule

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Motivation . . . 2
1.2 Ziele . . . 2
1.3 Umfang . . . 2

2 Ausgangslage **3**
2.1 Funktionale Anforderungen . . . 3
2.1.1 Ursprüngliches Ziel . . . 3
2.1.2 Rechner applikation . . . 3
2.2 Use cases . . . 3
2.2.1 Lange Summe . . . 3
2.2.2 Wiederholung und Konstante . . . 4
2.2.3 Auswertung eines Polynoms . . . 4
2.2.4 Komplexe Division . . . 4

3 Architektur **5**
3.1 Technologien . . . 5
3.2 Architektur . . . 5
3.3 Vue-Komponenten . . . 5

4 Benutzeroberfläche / User Experience **7**
4.1 Entscheidungen zur Benutzerschnittstelle . . . 7
4.1.1 Mathematische Körper . . . 7
4.1.2 Erweiterungen der Körper . . . 8
4.1.3 Takeover Knöpfe: Übernahme von Werten . . . 9
4.1.4 Polynom Eingabe . . . 10
4.1.5 Reverse Polish Notation . . . 11

5 Implementierung **13**
5.1 Körper und Polynomdarstellung im Code . . . 13
5.1.1 Körperklasse (Field) . . . 13
5.1.2 Operationen . . . 14
5.1.3 Ganzzahlen (`RingZ`) . . . 15

5.1.4 Körper der rationalen Zahlen (`FieldQ`) 16
5.1.5 Arithmetik in endlichen Körpern (`FieldFp`) 17
5.1.6 Polynomring (`ExtensionFields`) 18
5.1.7 Körpererweiterung mit Minimalpolynom (`ExtensionRxa`) 21
5.1.8 Minimalpolynom und Reduktion 22
5.1.9 Vordefinierte Minimalpolynome 23
5.2 Erweiterter euklidischer Algorithmus (`extendedGCD`) 23
5.3 Reverse Polish Notation 26
5.3.1 Implementierung 26
5.3.2 Funktionsweise 27
5.3.3 X, Y, Z, T 27
5.3.4 Methoden und Invariante 28

6 Erkenntnisse und Reflexion **30**

Bibliographie **31**

Abstract

Das Ziel der Arbeit war es, die Arithmetik in endlichen Körpern zu implementieren. Verwendet werden sollte diese dann in einer existierenden Web-Anwendung zum Gauss-Algorithmus (Gauss-Calculator). Die Implementation sollte dabei nicht nur den endlichen Körper $\mathbb{F}_p$ unterstützen, sondern auch Erweiterungen davon, wie Polynomringe $\mathbb{F}_p[x]$ und die Erweiterung um ein Minimalpolynom zu einem neuen Körper. Die existierende Arithmetik der Anwendung verwendete den double-Datentyp, eine Approximation des Körpers $\mathbb{Q}$. Diesen Körper sollte man neu über die Benutzeroberfläche wählen und konfigurieren können.

Um die Herausforderungen der Implementierung zu identifizieren, wurde zunächst eine detaillierte Analyse der Problemdomäne durchgeführt. Anschliessend wurde die Implementierung iterativ entwickelt, wobei ein Taschenrechner als erste Anwendung der Arithmetik implementiert wurde. Die Funktionalität des Rechners wurde schrittweise erweitert, um praktische Anwendungsfälle zu ermöglichen. Die Wahl der Reverse Polish Notation (RPN) für den Rechner ermöglicht eine effiziente Umsetzung der mathematischen Operationen. Die ursprünglich geplante Integration der entwickelten Lösung in den Gauss-Calculator wurde dem Auftraggeber überlassen.

Es gelang eine geeignete Abstraktion zu finden, mit der die Arithmetik und Zahlenformate der verschiedenen Körper und Ringe einfach mit der Benutzeroberfläche verbunden werden konnte. Diese besteht hautpsächlich aus einer Grundklasse, die verfügbare mathematische Operationen definiert. Implementiert werden diese für die einzelnen mathematischen Körper und Ringe in ihren eigenen Klassen.

Das Resulat dieser Arbeit ermöglicht es mit langen Ausdrücken in verschiedenen Körpern und Ringen zu rechnen und illustriert den Zusammenhang zwischen diesen.

Kapitel 1

Einleitung

1.1 Motivation

Mathematische Berechnungen in endlichen Körpern und Ringen sind grundlegend für Bereiche wie Kryptographie, Kodierungstheorie und Computer-Algebra. Die effiziente Implementierung dieser Operationen in Kombination mit einer benutzerfreundlichen Konfiguration stellt jedoch eine erhebliche Herausforderung dar.

1.2 Ziele

Das Hauptziel war es, die Arithmetik für endliche Körper ($\mathbb{F}_p$) und deren Erweiterungen, einschliesslich Polynomringen und Körpererweiterungen, in einem web-basierten Gauss-Calculator zu implementieren. Zusätzlich sollte die Benutzeroberfläche neu gestaltet werden, um Nutzern die dynamische Auswahl und Konfiguration mathematischer Strukturen zu ermöglichen.

1.3 Umfang

Die Implementierung umfasst:

- Arithmetik für $\mathbb{F}_p$.
- Arithmetik für Polynomringe ($\mathbb{F}_p[x]$).
- Erweiterungen von $\mathbb{F}_p[x]$ durch Minimalpolynome.
- Einen Taschenrechner mit Reverse Polish Notation für praktische Anwendungen.

Kapitel 2

Ausgangslage

2.1 Funktionale Anforderungen

2.1.1 Ursprüngliches Ziel

Das Ziel der Arbeit ist die Implementierung der Arithmetik in einem endlichen Körper. Diese Implemenation soll in einer existierenden Web-Applikation für den Gauss-Algorithmus eingebunden werden. Damit soll mit dem Gauss-Algorithmus auch in anderen Körpern als nur im bisherigen Körper $\mathbb{Q}$ gerechnet werden können.

2.1.2 Rechner applikation

Durch die iterative Entwicklung der arithmetischen Implementierung wurde ein Rechner erstellt. Dieser wurde später erweitert, um praktische Anwendungsfälle zu unterstützen. Die Integration der finalen Arithmetik-Implementierung in die ursprüngliche Applikation wurde dem Auftraggeber überlassen.

2.2 Use cases

Diese Use Cases dienen als Basis für die Weiterentwicklung des Rechners, mit dem sie nur sehr aufwendig, mit separatem Notieren von Zwischenresultaten, zu erfüllen waren.

2.2.1 Lange Summe

Für die Berechnung eines Matrizenproduktes in einem endlichen Körper muss man Zeilen mit Spalten multiplizieren, also Ausdrücke der Form

$$a_{21} \cdot b_{13} + a_{22} \cdot b_{23} + a_{23} \cdot b_{33}$$

berechnen (Element 2,3 der Produktmatrix AB). Das Problem dabei ist, dass man sich das Resultat des ersten Produktes merken muss, das zweite Produkt ausmultiplizieren muss und erst dann die Summe bilden kann.

2.2.2 Wiederholung und Konstante

Z. B. in der Signalverarbeitung wird oft eine Serie von Datenpunkten B mit einer Konstanten gerechnet. Sie verwendet einen Accumulator (A) und berechnet mit jedem neuen Datenpunkt B ein Produkt $B \cdot C$ (mit einer vorgegebenen Konstanten C) und addiert das Produkt zu A. Für Formeln wie

$$A := A + B \cdot C$$

muss A gemerkt werden, das Produkt ausgewertet und dann zu A hinzuaddiert werden.

2.2.3 Auswertung eines Polynoms

Man möchte man in der Lage sein, ein Polynom auszuwerten. Man möchte also in einem Polnom wie

$$a \cdot X^2 + b \cdot X + c$$

für X einen Wert d einsetzen.

2.2.4 Komplexe Division

Das Besondere an einem Körper ist, dass man immer dividieren kann. Man möchte also auch rationale Funktionen wie

$$f(x) = \frac{a \cdot x + b}{c \cdot x + d}$$

auswerten. Schon in diesem Fall ist man wieder mit dem Problem konfrontiert, dass man sich den Wert des Zählers merken und den Nenner berechnen muss, bevor man die Division ausführen kann.

Kapitel 3

Architektur

3.1 Technologien

Die bereits bestehende Applikation wurde mit JavaScript und Vue.js entwickelt. Um die Integration und Wartbarkeit der zu entwickelnden Funktionalität zu gewährleisten, wurden dieselben Technologien gewählt. Zusätzlich wurde automatisiertes Testing mit Gitlab CI/CD und der JavaScript Bibliothek Jasmine benutzt.

3.2 Architektur

Die Implementation der Arithmetik sollte, um Testing zu ermöglichen, getrennt von der View Logik erfolgen. Es wurde deshalb ein Model-View-Controller-Ansatz (MVC) gewählt, der mit Vue.js gut umsetzbar ist. Für die Domänenlogik (das Modell) wurden separate Körper-Klassen entwickelt. Diese bilden die arithmetischen Körper (und Ringe) ab. Durch den JavaScript Teil von Vue.js wird der Controller realisiert. Dieser stellt sicher, dass das Format zwischen Eingabefeldern und Körper-Klassen übereinstimmt. Das template in Vue.js dient als View-Teil. Das Framework selber übernimmt dabei auch Funktionen des Controllers.

3.3 Vue-Komponenten

Das ursprüngliche Ziel der Arbeit war die Integration in eine existierende Gauss-Algorithmus-Applikation. Diese nutzte standardisierte HTML-Eingaben zur Erfassung der Operatoren. Um den bestehenden Code möglichst wenig anpassen zu müssen, haben wir diese Eingaben durch Vue-Komponenten ersetzt. Für verschiedene Körper, wie beispielsweise Polynome, wurden Eingabefelder entwickelt, die die entsprechenden Zahlen oder Koeffizienten formatiert zurückgeben. Die umgebende Hauptkomponente empfängt diese Werte und nutzt sie für die Berechnungen. Je nach gewähltem Körper oder Körpererweiterung ist es notwendig, zusätzliche Konfigurationen vorzunehmen.

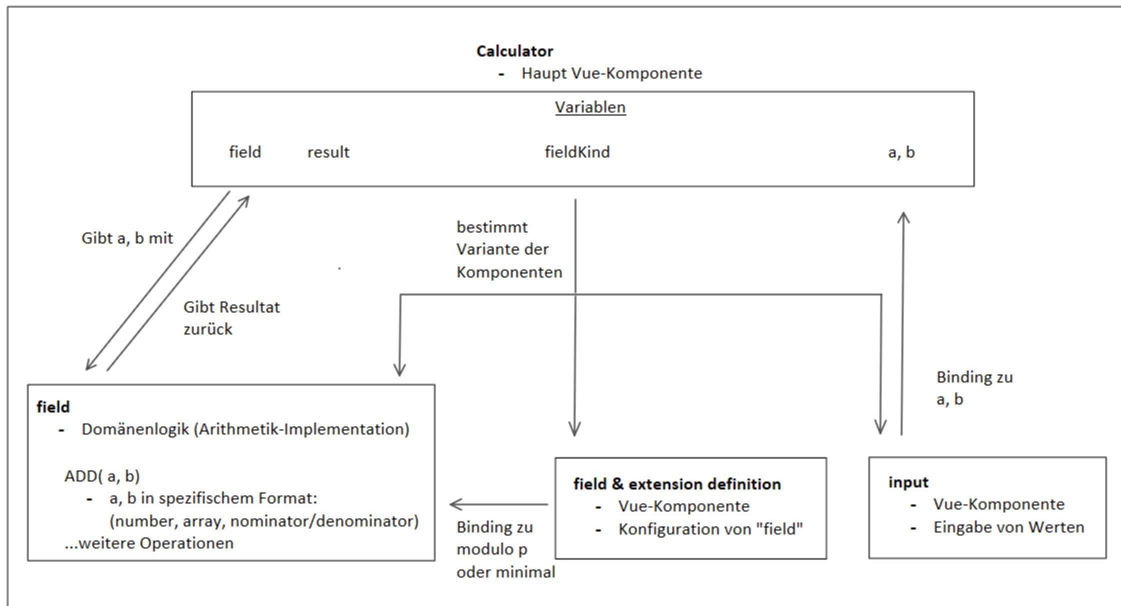


Abbildung 3.1 – Übersicht der Vue-Komponenten der Implementierung und deren Interaktion.

Es handelt sich also total um 4 Komponenten:

- Hauptkomponente (Calculator)
- Eingabefelder (input)
- Körperdefinition (field definition)
- Erweiterungsdefinition (extension definition)

Wie in der Abbildung 3.1 zu sehen ist, definiert die Hautptkomponente welche Komponenten angezeigt werden über die Auswahl von *fieldKind* in der Benutzeroberfläche. Sie nimmt die Werte für a und b von der *input* Komponente entgegen und gibt diese an die Arithmetik-Implemenation (*field*) weiter, die verantwortlich für die Logik ist. Die *field/extension definition* Komponenten dienen der Konfiguration der gewählten Arithmetischen Implemenation. Beispielsweise die Wahl eines Minimalpolynomes *minimal* oder die Eingabe des Modulos p. Sie haben ein indirektes Binding auf die internen Werte von *field* über die Hauptkomponente.

Kapitel 4

Benutzeroberfläche / User Experience

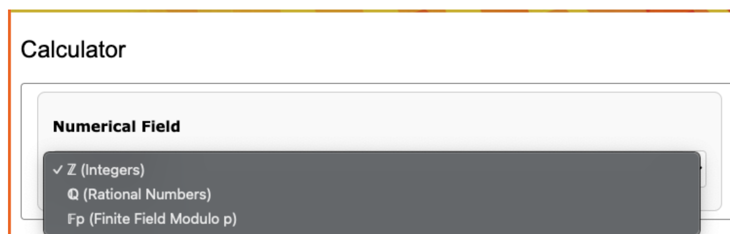
4.1 Entscheidungen zur Benutzerschnittstelle

In diesem Abschnitt werden die Entscheidungen bezüglich der Benutzeroberfläche und -erfahrung dargestellt. Dies umfasst die Gestaltung der Eingabemethoden, die Bereitstellung von Erweiterungen sowie die allgemeinen Mechanismen zur Verwendung der verschiedenen mathematischen Körper und ihrer Erweiterungen.

4.1.1 Mathematische Körper

Die Zahlenbereiche, die in der Benutzeroberfläche dargestellt werden, sind:

- **Ganzzahlen ($\mathbb{Z}$)**: Der Ring der Ganzzahlen ist die Standard-Auswahl und erlaubt die Eingabe ganzer Zahlen für arithmetische Operationen. Der Benutzer hat direkten Zugriff auf die vier Grundoperationen: Addition, Subtraktion, Multiplikation und Division (Division mit Rest).
- **Rationale Zahlen ($\mathbb{Q}$)**: Für die rationalen Zahlen können Benutzer Werte als Brüche eingeben, wobei sowohl Zähler als auch Nenner anzugeben sind. Dies gibt den Benutzern eine präzisere Möglichkeit, mit rationalen Zahlen zu rechnen und gleichzeitig alle wesentlichen arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division) auszuführen.
- **Endliche Körper ($\mathbb{F}_p$)**: Die endlichen Körper ($\mathbb{F}_p$) bieten die Möglichkeit der modularen Arithmetik. Der Benutzer kann eine Primzahl p eingeben, um die Operationen in einem endlichen Körper zu definieren. Alle arithmetischen Operationen erfolgen dann modulo p. Dies ist besonders relevant für kryptographische Berechnungen und die Zahlentheorie.

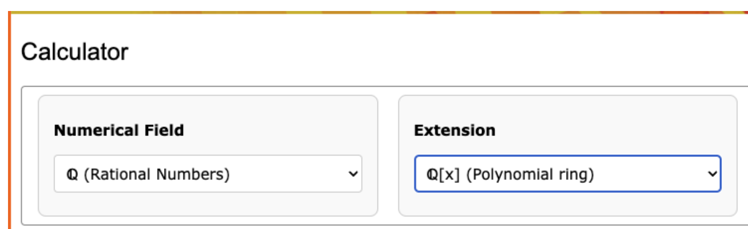


4.1.2 Erweiterungen der Körper

Neben den Körpern $\mathbb{Q}, \mathbb{F}_p$ und dem Ring $\mathbb{Z}$, gibt es verschiedene Erweiterungen, die durch den Benutzer definiert werden können. Diese Erweiterungen bieten erweiterte Operationen für spezielle Anwendungen.

Erweiterung von $\mathbb{Q}$ (Polynomring $\mathbb{Q}[x]$)

Für die rationalen Zahlen $\mathbb{Q}$ wurde die polynomische Erweiterung $\mathbb{Q}[x]$ implementiert. Diese Erweiterung erlaubt die Durchführung von Operationen auf rationalen Polynomen. Benutzer können Polynome als Listen von Koeffizienten eingeben, wobei die Implementierung sicherstellt, dass alle arithmetischen Operationen auf Polynomen ausgeführt werden können (Addition, Subtraktion, Multiplikation, Division).



Erweiterung von $\mathbb{F}_p$

Für den endlichen Körper $\mathbb{F}_p$ gibt es zwei Arten von Erweiterungen, die besonders für Anwendungen in der Kryptographie von Bedeutung sind.

Polynomring $\mathbb{F}_p[x]$ Die erste Erweiterung ist eine polynomische Erweiterung, das pendent zum Polynomring $\mathbb{Q}[x]$. Diese Erweiterung erlaubt es dem Benutzer, mit Polynomen in einem endlichen Körper zu arbeiten.

Erweiterung mit Minimalpolynom $\mathbb{F}_p[x]/(m)$ Die zweite Erweiterung ist die Möglichkeit der Körpererweiterung um ein Minimalpolynom. Es muss ein Minimalpolynom definiert werden, das verwendet wird, um Operationen in einer Erweiterung des Körpers zu ermöglichen. Der Benutzer kann ein vordefiniertes Minimalpolynom auswählen oder ein eigenes definieren, um die Erweiterung des Körpers zu steuern. Das führt dazu, dass

Calculator

Numerical Field

𝔽p (Finite Field Modulo p)

Field definition

Modulo P:

5

Extension

𝔽p[x] (Polynomial ring)

der Polynomring über $\mathbb{F}_p$ zu einem Körper wird. So kann beispielsweise im Körper $\mathbb{F}_{2^8}$ gerechnet werden, der z. B. für den Advanced Encryption Standard AES benutzt wird.

Calculator - $\mathbb{F}_p$ [x] / (m)

Numerical Field

$\mathbb{F}_p$ (Finite Field Modulo p)

Field definition

Modulo p:

5

Extension

$\mathbb{F}_p$[x] / (m) (Extension field)

Minimal Polynomial (m)

deg(2): x^2 + x + 1

4.1.3 Takeover Knöpfe: Übernahme von Werten

Eine neue Funktionalität, die in die Benutzeroberfläche eingeführt wurde, ist die Verwendung von sogenannten 'Takeover'-Knöpfe. Diese Knöpfe ermöglichen es dem Benutzer, das Resultat einer Operation schnell als neuen Eingabewert für eine weitere Berechnung zu übernehmen.

Die beiden Takeover-Knöpfe sind:

- **Takeover to A:** Der Knopf, mit dem das aktuelle Resultat als Eingabewert für das Polynom a übernommen wird.

- **Takeover to B:** Der Knopf, mit dem das aktuelle Resultat als Eingabewert für das Polynom b übernommen wird.

Die beiden Knöpfe befinden sich jeweils in der Nähe des Ergebnisbereichs, sodass der Benutzer nach der Durchführung einer Berechnung die Möglichkeit hat, das Resultat einfach und schnell in die Operandfelder a oder b zu übernehmen, ohne die Werte manuell übertragen zu müssen.

Diese Entscheidung wurde getroffen, um den Arbeitsfluss zu vereinfachen und dem Benutzer die Möglichkeit zu geben, leicht weitere Operationen mit dem Resultat durchzuführen, insbesondere im Kontext von Polynomoperationen, wo es sinnvoll ist, Ergebnisse erneut zu verwenden.

4.1.4 Polynom Eingabe

Polynome können in Stellenform eingegeben werden. Dadurch wird dem Benutzer die Eingabe des Grades erspart. Durch Tabbing kann durch die Koeffizienten navigiert werden, was die Eingabe von links nach rechts komfortabel ermöglicht. Der Grad lässt sich durch einen Knopf erhöhen. Dieser fügt ein neues Eingabefeld hinzu und fokussiert dieses.

4.1.5 Reverse Polish Notation

Um komplexere Ausdrücke mit mehr als zwei Zahlen einzugeben, haben wir uns für eine Implementierung eines Reverse Polish Notation (RPN) Rechners entschieden. Da wir die Eingabe von Zahlen in verschiedene Körpern (vor allem Polynome) über speziell dafür implementierte Eingabefelder gelöst haben, ist dies in der Implementierung einfacher als Ausdrücke zu parsen. Der Ansatz unterscheidet sich darin, dass wir nicht von einem Text den mathematischen Parse-Tree generieren, sondern ihn über spezielle Eingabefelder direkt erhalten. Ebenfalls ist für den Benutzer die Eingabe von langen Ausdrücken mit Polynomen einfacher als über eine Text-Eingabe.

Die Wahl der RPN erlaubt eine effiziente und intuitive Verarbeitung von mathematischen Ausdrücken. Die Benutzeroberfläche bietet eine einfache Möglichkeit, Werte und Operatoren einzugeben. Zudem werden durch die Verwendung des Stacks Zwischenergebnisse automatisch verwaltet, wodurch sich die Fehleranfälligkeit reduziert.

Um die intuitive Benutzbarkeit des zuerst implementierten Rechners mit zwei Operatoren für einfache Operationen zu bewahren, haben wir uns entschieden, zwischen den zwei Anwendungsfällen zu unterscheiden. Das erlaubt es, die beste Lösung für beide ohne Kompromisse anzubieten. Der Wechsel zwischen den Anwendungsfällen ist über einen Modus-Schalter implementiert (unten rechts zu sehen).

Anstelle der normalen Eingabefelder wird der Stack angezeigt. Dabei werden die Werte T, Z, Y und X von oben nach unten im Stack angezeigt. Der unterste, aktuelle Wert X ist editierbar, wie bei den vorher benutzten Inputs.

Die arithmetischen und Stack Operationen werden unter dem Stack auf separaten Zeilen angezeigt.

Anzahl Elemente

Grundsätzlich würde ein Rechner mit Reverse Polish Notation mit unendlich vielen Elementen auf dem Stack bzw. Anzahl Registern funktionieren. Damit der Benutzer jedoch

jederzeit den Überblick über alle Elemente behalten kann, ist die Limitierung auf eine fixe Anzahl notwendig. Dies ist vor allem bei der Rotations-Operation wichtig, bei der Werte von Unten nach oben auf dem Stack wandern.

Die Limitierung der Anzahl Elemente erlaubt es auch, dass nach Operationen, bei denen sich die Anzahl der Elemente reduzieren würde, das oberste Element `T` dupliziert wird. In Kombination mit der Rotate-Funktion erlaubt dies eine Konstante oben auf dem Stack zu behalten, mit der stets weitergerechnet werden kann.

Kapitel 5

Implementierung

5.1 Körper und Polynomdarstellung im Code

Die Klassen `Field`, `RingZ`, `FieldFp`, `FieldQ`, `ExtensionFields` und `ExtensionRxa` repräsentieren verschiedene mathematische Körper und Ringe und deren Arithmetik im JavaScript-Code.

5.1.1 Körperklasse (Field)

Die Basisklasse `Field` definiert die grundlegenden Operationen Addition, Subtraktion, Multiplikation und Division. Jede abgeleitete Klasse (z.B. `RingZ`, `FieldQ`) muss diese Operationen implementieren. Dies ist notwendig, weil die Rechenregeln für verschiedene Körper unterschiedlich sind. Zusätzlich sind die Hilfsmethoden `inverse`, `negate` und `numberEquals` implementiert, die spezifische Eigenschaften der jeweiligen Zahlkörper unterstützen.

```
1  class Field {
2      constructor(kind, needsDefinition, isDefined) {
3          // ...
4      }
5
6      add(a, b) {
7          throw new Error("add() must be implemented by subclass");
8      }
9      sub(a, b) {
10         throw new Error("sub() must be implemented by subclass");
11     }
12
13     mul(a, b) {
14         throw new Error("mul() must be implemented by subclass");
15     }
16
17     div(a, b) {
18         throw new Error("div() must be implemented by subclass");
19     }
```

```
20
21      negate(a) {
22          const res = this.sub(this.zero, a)[0];
23          return [res];
24      }
25
26      inverse(a) {
27          // Default: inverse = div(1, a)
28          if (this.numberEquals(a, this.zero)) {
29              return NaN;
30          }
31          const res = this.div(this.padOperator(1), a)[0];
32          if (res === null || (typeof res === 'number' && Number.isNaN(res)
               )) {
33              return NaN;
34          }
35          return res;
36      }
37      // weitere abstrakte Methoden...
38  }
```

Methode `negate(a)`: Die Methode `negate` berechnet das **additive Inverse** eines Wertes a. Sie wird mit der Subtraktionsmethode implementiert und ergibt:

$$\text{negate}(a) = 0 - a$$

Damit wird das Vorzeichen von a umgekehrt. Diese Methode ist insbesondere in Ringen wie $\mathbb{Z}$ oder $\mathbb{F}_p[x]$ relevant.

Methode `inverse(a)`: Die Methode `inverse` berechnet das **multiplikative Inverse** eines Wertes a. Dies ist ein Wert b, für den gilt:

$$a \cdot b = 1$$

Die Methode wird mithilfe der Division (`div`) implementiert, wobei 1 durch a geteilt wird. Es gilt:

$$\text{inverse}(a) = \text{div}(1, a)$$

Falls a gleich 0 ist oder kein multiplikatives Inverses existiert, gibt die Methode `NaN` zurück. Dies ist vor allem in Körpern wie $\mathbb{F}_p$ wichtig, wo jedes Element (ausser 0) ein Inverses besitzt. In Ringen wie $\mathbb{Z}$ oder $\mathbb{F}_p[x]$ ist das Inverse nur für bestimmte Elemente definiert.

5.1.2 Operationen

Jede Rechenoperation gibt ein Tupel von [*Resultat*, *Schritte*, *Rest*] zurück. Die Schritte sind dabei eine Veranschaulichung, wie das Resulat berechnet wurde. Der Rest ist nur bei der Division mit Rest in Ringen relevant.

Zum Beispiel: Die Addition zweier Ganzzahlen a und b im Ring $\mathbb{Z}$ erfolgt einfach durch ihre Summe, was in den Schritten (*steps*) erläutert ist. Einen Rest gibt es dabei nie.

```
1     add(a, b) {
2         const result = a + b;
3         const steps = `Addiere die Zahlen: ${a} + ${b} = ${result}`;
4         return [result, steps];
5         }
```

Alle binären Operationen werden aufgerufen und ihre Resultate dargestellt. Dank dieser Abstraktion können alle Knöpfe für die Operationen dieselbe Methode verwenden.

```
1     operationNormalMode(operation) {
2         [this.result, this.steps, this.remainder] = this.operationField[
              operation](this.a, this.b);
3         if(this.result !== null){...}
4     ...}
```

Ein *null* wert im Resultat wird hier nach Konvention als Fehler zurückgegeben, wobei als *steps* eine Beschreibung des Fehlers angezeigt wird. Dies zum Beispiel, falls kein Inverses gefunden wird bei der Division im endlichen Körper $\mathbb{F}_p$.

5.1.3 Ganzzahlen (`RingZ`)

Die Klasse `RingZ` implementiert die Ganzzahlen $\mathbb{Z}$. Für Addition und Subtraktion können direkt die entsprechenden JavaScript Operationen $+$ und $-$ verwendet werden.

Division

Die Division der Ganzzahlen erfolgt mit Rest. Vielfach wird diese mit z. B. `Math.floor(a / b)` implementiert. Dies ist jedoch für negative Zahlen nicht korrekt, da die Funktion `Math.floor` in JavaScript gegen $-\infty$ rundet. Daher wurde folgende Implementierung gewählt:

```
1 const result = (b < 0)
2     ? (a >= 0)
3         ? Math.trunc(a / b)
4         : Math.ceil(a / b)
5     : Math.floor(a / b);
6 const remainder = a - b * result;
```

Falls der Divisor b negativ ist und a positiv, muss auf 0 gerundet werden (`Math.trunc`). Zum Beispiel ergibt $10/(-3)$:

$$10/(-3) = -3 \quad \text{Rest} \quad 1.$$

Ist der Divisor b sowie a negativ, muss hingegen aufgerundet werden (`Math.ceil`):

$$-10/(-3) = 4 \quad \text{Rest} \quad 2.$$

Inverses

Bei den Ganzzahlen ist nur das Inverse von 1 und −1 definiert. Für diese beiden Werte wird die Zahl selbst zurückgegeben, wie in der Implementierung ersichtlich ist.

```
1       inverse(a) {
2           if(a === 1 || a === -1) return a;
3           else return NaN
4       }
```

Für jegliche andere Zahlen wird der Fehlerwert NaN zurückgegeben.

5.1.4 Körper der rationalen Zahlen (`FieldQ`)

Die Klasse `FieldQ` implementiert die Arithmetik für rationale Zahlen, dargestellt als Brüche. Die Methoden behandeln Brüche als Arrays, wobei das erste Element den Zähler und das zweite den Nenner darstellt.

Zum Beispiel: Die Addition zweier Brüche $\frac{a}{b}$ und $\frac{c}{d}$ wird wie folgt berechnet:

$$\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d}.$$

Dies ist in der *add* Methode der Klasse (unten) als *numerator* und *denominator* implementiert.

```
1   add(a, b) {
2       const [numA, denA] = a, [numB, denB] = b;
3       const numerator = numA * denB + denA * numB;
4       const denominator = denA * denB;
5       const [simpleNumerator, simpleDenominator] = this.simplify(numerator,
            denominator);
6       return [[simpleNumerator, simpleDenominator], steps];
7   }
```

Zusätzlich wird der Bruch mithilfe der Methode `simplify(numerator, denominator)` auf seinen kleinsten Nenner gekürzt. Dabei wird der grösste gemeinsame Teiler (ggT) verwendet, um das Ergebnis zu vereinfachen. Der ggT selbst wird mit der Methode `gcd(a, b)` berechnet, die den grössten gemeinsamen Teiler zweier Zahlen liefert.

```
1   simplify(numerator, denominator) {
2       const gcd = this.gcd(numerator, denominator);
3       return [numerator / gcd, denominator / gcd];
4   }
5
6   gcd(a, b) {
7       if (b === 0) {
8           return a;
```

```
9       }
10      return this.gcd(b, a % b);
11  }
```

5.1.5 Arithmetik in endlichen Körpern (FieldFp)

Die Klasse `FieldFp` implementiert die Arithmetik in endlichen Körpern, die oft in der Zahlentheorie und Kryptografie verwendet wird. Bei den hier implementierten Operationen handelt es sich um Arithmetik modulo einer Primzahl p. Die Implementierung basiert auf den Prinzipien der endlichen Körper, wie in [Mül23, S.211, Satz 5.11] beschrieben.

Addition im Körper $\mathbb{F}_p$

Die Addition im endlichen Körper $\mathbb{F}_p$ von zwei Zahlen a und b wird durch die Formel

$$(a + b) \mod p$$

berechnet [Mül23, Satz 2.1, S. 56]. In der Körperklasse wird diese Rechnung direkt so als Implementation verwendet und mit Schritten erklärt:

```
1  add(a, b) {
2      const result = (a + b) % this.p;
3      const steps = `
4          Addiere die Zahlen: ${a} + ${b} = ${a + b}
5          Wende modulo ${this.p} an: (${a + b}) % ${this.p} = ${result}
6      `;
7      return [result, steps];
8  }
```

Subtraktion im Körper $\mathbb{F}_p$

Die Subtraktion von zwei Zahlen im Körper $\mathbb{F}_p$ wird anhand der Formel

$$(a - b) \mod p$$

durchgeführt [Mül23, Satz 2.1, S. 57]. Das Ergebnis wird durch die Verwendung der Modulo-Operation sichergestellt.

Um die Modulo-Operation bei der Subtraktion zu implementieren, wird hier noch einmal p addiert, damit das Resultat sicher positiv ist. Dies ist notwendig, weil in JavaScript beispielsweise `-3 % 5 === -3` gilt
Das gewünschte positive Ergebnis kann erreicht werden, indem p addiert wird.

```
1  sub(a, b) {
2      const result = (a - b + this.p) % this.p;
3      const steps = `
4          Subtrahiere die Zahlen: ${a} - ${b} = ${a - b}
```

```
5           Wende modulo ${this.p} an: (${a - b + this.p}) % ${this.p} = ${
                result}
6       `;
7       return [result, steps];
8   }
```

Multiplikation im Körper $\mathbb{F}_p$

Die Multiplikation von zwei Zahlen im Körper $\mathbb{F}_p$ wird ebenfalls durch die Verwendung der Modulo-Operation sichergestellt. Die Rechnung ist dabei ähnlich wie schon für die Addition und Subtraktion:

$$(a \cdot b) \mod p$$

[Mül23, Satz 2.1, S. 58].

```
1   mul(a, b) {
2       const result = (a * b) % this.p;
3       const steps = `...`;
4       return [result, steps];
5   }
```

Division im Körper $\mathbb{F}_p$

Für die Division im endlichen Körper wird die modulare Inverse verwendet. Dies bedeutet, dass man statt durch eine Zahl zu dividieren, mit ihrem inversen Wert multipliziert. Also

$$\frac{a}{b} = (a \cdot b^{-1}) \mod p,$$

wobei b^{-1} die modulare Inverse von b im Körper $\mathbb{F}_p$ ist. Diese Inverse kann mit dem erweiterten euklidischen Algorithmus berechnet werden [Mül23, Satz 2.3, S. 60].

```
1   div(a, b) {
2       const invB = this.inverse(this.padOperator(b));
3       const result = (a * invB) % this.p;
4       const steps = `...`;
5       return [result, steps];
6   }
```

Die Methode *inverse* ruft hier den euklidischen Algorithmus auf und gibt das Inverse des Divisors zurück. Die Funktion *padOperator* sorgt dafür, dass b ein Element des definierten Körpers ist. Also $0 \leq b < p$.

5.1.6 Polynomring (`ExtensionFields`)

Datentyp

Der Polynomring $\mathbb{R}[x]$ ist in `ExtensionFields` implementiert. Der Körper der Koeffizienten ist als Attribut *field* definiert und wird verwendet für Operationen auf diesen.

Dadurch kann dieselbe Implementierung für Polynome über $\mathbb{F}_p$, sowie $\mathbb{Q}$ verwendet werden.

Polynome sind als Array implementiert. Die Elemente repräsentieren die Koeffizienten des Polynoms. Die Position k im Array bestimmt den Grad des entsprechenden Koeffizienten, wobei gilt:

$$\text{Grad} = (\text{Länge des Arrays}) - k - 1.$$

An der Position 0 des Arrays steht somit der Koeffizient des höchsten Grades n, wobei n der Grad des Polynoms ist.

Beispiel: Ein Polynom $p(x) = 3x^2+2x+1$ wird durch seinen Array der Koeffizienten

$$[3, 2, 1],$$

dargestellt. Dabei steht der erste Eintrag 3 für den Koeffizienten von x^2, der zweite Eintrag 2 für den Koeffizienten von x und der letzte Eintrag 1 für die Konstante.

Polynomaddition

Die Addition zweier Polynome $p_1(x)$ und $p_2(x)$ wird durch koeffizientenweise Addition realisiert. Beide Polynome werden zunächst auf die gleiche Länge gebracht, indem sie mit Nullen aufgefüllt werden.

```
1 add(arr1, arr2) {
2     const maxLength = Math.max(arr1.length, arr2.length);
3     const poly1 = this.parseArrayToMap(arr1, maxLength);
4     const poly2 = this.parseArrayToMap(arr2, maxLength);
5
6     for (let i = 0; i < maxLength; i++) {
7         result.set(i, (poly1.get(i) || 0) + (poly2.get(i) || 0));
8     }
9 }
```

Zurest wird in der Implementation die maximale Länge *maxLength* bestimmt. Danach werden die Polynome mit der Methode *parseArrayToMap* mit Nullen aufgefüllt.

Das Ergebnis ist die Summe der entsprechenden Koeffizienten an jeder Potenz von x. Die Summe kann nun in einem Loop berechnet werden, da beide Polynom-Maps gleich viele Einträge haben. Diese Implementierung basiert auf den Prinzipien der Polynomarithmetik, wie in [Mül23, Satz 5.5, S.203] beschrieben.

Polynomsubtraktion

Die Subtraktion funktioniert analog zur Addition, indem die Koeffizienten von $p_2(x)$ von denen von $p_1(x)$ subtrahiert werden. Wieder werden die Polynome zunächst gleich lang gemacht.

```
sub(arr1, arr2) {
    const maxLength = Math.max(arr1.length, arr2.length);
    const poly1 = this.parseArrayToMap(arr1, maxLength);
    const poly2 = this.parseArrayToMap(arr2, maxLength);

    for (let i = 0; i < maxLength; i++) {
        result.set(i, (poly1.get(i) || 0) - (poly2.get(i) || 0));
    }
}
```

Es werden die gleichen Methoden wie bei der Addition wiederverwendet. Lediglich die Operation in der `for` Schleife unterscheidet sich. Hier werden die Koeffizienten von $p_2(x)$ von denen von $p_1(x)$ subtrahiert. Die Methodik für Polynomarithmetik wurde inspiriert von den Ausführungen in [Mül23, Satz 5.5, S.203].

Polynommultiplikation

Die Multiplikation zweier Polynome $p_1(x)$ und $p_2(x)$ wird realisiert, indem jedes Monom aus $p_1(x)$ mit jedem Monom aus $p_2(x)$ multipliziert wird. Der Grad der entstehenden Terme wird addiert.

```
mul(poly1, poly2) {
    for (const [degA, coefA] of poly1.entries()) {
        for (const [degB, coefB] of poly2.entries()) {
            const newDegree = degA + degB;
            const newCoefficient = coefA * coefB;

            if (result.has(newDegree)) {
                result.set(newDegree, result.get(newDegree) +
                    newCoefficient);
            } else {
                result.set(newDegree, newCoefficient);
            }
        }
    }
}
```

Das Ergebnis ist ein neues Polynom, das durch die Summe der Produkte aller möglichen Kombinationen der Terme von p_1 und p_2 entsteht. Die Implementierung basiert auf den Regeln der Polynommultiplikation, wie sie in [Mül23, Satz 5.5, S.203] beschrieben sind.

Division

Die Division von Divisor $= a$, Dividend $= b$ nimmt zwei Polynome entgegen und gibt Quotient und Rest zurück. Zuerst werden führende Nullstellen von den Polynomen entfernt. Dadurch lässt sich der Grad eines Polynoms über die Länge des Arrays bestimmen: $\deg(a) = a.length - 1$. Ist $\deg(a) > \deg(b)$, wird Quotient $= 0$, Rest $= a$ zurückgegeben.

Algorithmus

Implementiert wird die Division mit der Tableaumethode, wie in [Mül23, S.211, Satz 5.11] beschrieben.
Das Tableau t wird als 2D Array repräentiert mit Array[*Zeile*][*Spalte*]. Die linke Seite des Tableaus ist dabei in $Spalte < \deg(a)$, die rechte Seite in $Spalte \geq \deg(a)$.
Das Tableau wird in 2 Schritten aufgefüllt:

1. Zuerst werden `tableauLength = (a.length - b.length)+ 1` Zeilen von b in einem Loop mit entsprechendem Padding von Nullen gefüllt.
   Die linke Seite hat *index* führende Nulleinträge, gefolgt von b. Auf der rechten Seite steht -1, dann `tableauLength - index - 1` Nulleinträge. Total ergibt dies `tableauLength - 1` Nulleinträge auf beiden Seiten.

2. Die letzte Zeile des Tableaus wird mit a gefüllt, gefolgt von Nulleinträgen.

```
1     const paddingNeeded = tableauLength - index - 1;
2     const paddedB = [...Array(index).fill(0), ...b, ...Array(
          paddingNeeded).fill(0)]
3     const rightSide = [...Array(index).fill(0), -1, ...Array(
          paddingNeeded).fill(0)]
4     return [...paddedB, ...rightSide]
```

Auf das Tableau wird dann die Forwärtsreduktion angewandt in einer Schleife von `index = [0 - tableauLength]` und mit `pivot = tableau[index][index]`. Also über die Diagonale des Tableaus.

Schlussendlich kann das Resultat der letzten Zeile des Tableaus entnommen werden:

```
1     const remainder = resultRow.slice(0, a.length);
2     const quotient = resultRow.slice(a.length);
```

5.1.7 Körpererweiterung mit Minimalpolynom (ExtensionRxa)

Die Klasse `ExtensionRxa` erweitert die bestehende Klasse `ExtensionFields`, die Polynomringe repräsentiert, um zusätzliche Funktionalitäten. Im Wesentlichen erweitert `ExtensionRxa` den Polynomring durch die Einführung eines minimalen Polynoms. Diese Erweiterung erlaubt es, Arithmetik innerhalb eines sogenannten Erweiterungskörper durchzuführen, bei dem nach jeder Multiplikation eine Reduktion durch das minimale Polynom erfolgt.

Klassendefinition und Eigenschaften

Die Klasse `ExtensionRxa` implementiert erweiterte Polynomoperationen und führt dabei Reduktionen durch ein Minimalpolynom *minimal* durch. Das Minimalpolynom wird entweder vom Benutzer festgelegt oder aus einer vorgegebenen Liste *predefinedMinimals* ausgewählt.

```
class ExtensionRxa extends ExtensionFields {

    static extensionName = 'Rxa';
    minimal;

    predefinedMinimals = [
        {id: 3, value: [1, 1, 1], label: 'deg(2): x^2 + x + 1'},
        {id: 4, value: [1, 0, 1, 1], label: 'deg(3): x^3 + x + 1'},
        {id: 5, value: [1, 1, 0, 1], label: 'deg(3): x^3 + x^2 + 1'},
        {id: 8, value: [1, 0, 0, 0, 1, 1, 0, 1, 1], label: 'deg(8): x^8 +
             x^4 + x^3 + x + 1'}
    ]
    ...
}
```

Die Klasse `ExtensionRxa` erbt von `ExtensionFields`, um alle grundlegenden polynomiellen Operationen wie Addition, Subtraktion, Multiplikation und Division zu übernehmen. Eine wichtige Eigenschaft von `ExtensionRxa` ist die Möglichkeit, ein Minimalpolynom anzugeben, das zur Reduktion der Ergebnisse von Multiplikationsoperationen verwendet wird.

5.1.8 Minimalpolynom und Reduktion

Ein zentraler Bestandteil der Klasse `ExtensionRxa` ist das Konzept des Minimalpolynoms. Es handelt sich dabei um ein Polynom, das zur Definition der Erweiterung verwendet wird. Nach jeder Multiplikation wird das resultierende Polynom durch das Minimalpolynom reduziert. Diese Vorgehensweise stellt sicher, dass das Ergebnis innerhalb der durch das Minimalpolynom definierten algebraischen Struktur bleibt.

Reduktion eines Polynoms

Die Methode zur Reduktion eines Polynoms nach einer Multiplikation verwendet die Division des Polynoms durch das Minimalpolynom und nimmt den Rest als Ergebnis. Dies entspricht der Definition eines Polynomrings, der durch ein irreduzibles Minimalpolynom erzeugt wird.

```
reducePolynomial(polynomial) {
    const [quotient, steps, remainder] = this.div(polynomial, this.
       minimal);
    return this.parseArrayToMap(remainder);
}
```

Der Rückgabewert ist das reduzierte Polynom, dargestellt als Map. Die Methode `div()` ist eine Division, die den Quotienten und den Rest als Tupel zurückgibt. Für die Reduktion wird der Rest verwendet, um sicherzustellen, dass das Ergebnis innerhalb des durch das Minimalpolynom definierten Rings bleibt.

Multiplikation mit Reduktion

In der Klasse `ExtensionRxa` wird die Multiplikation durch die Methode `mul(poly1, poly2)` überschrieben, um sicherzustellen, dass jedes Produkt nach der Berechnung durch das Minimalpolynom reduziert wird.

```
1 mul(poly1, poly2) {
2     const [product, steps] = super.mul(poly1, poly2);
3     const reducedPolynomial = this.reducePolynomial(product);
4     return [reducedPolynomial, `${steps}\nReduced polynomial: ${this.
        mapToPolynomialString(reducedPolynomial)}`];
5 }
```

Diese Methode ruft die Multiplikationsmethode der Basisklasse (`ExtensionFields`) auf, um das Produkt der beiden Polynome zu berechnen. Anschliessend wird das resultierende Produkt durch das Minimalpolynom reduziert, um das endgültige Ergebnis zu erhalten. Der Rückgabewert enthält sowohl das reduzierte Polynom als auch die Beschreibung der durchgeführten Schritte.

5.1.9 Vordefinierte Minimalpolynome

Die Klasse `ExtensionRxa` bietet eine Reihe von vordefinierten Minimalpolynomen, aus denen der Benutzer auswählen kann. Diese Minimalpolynome ermöglichen eine einfache Erweiterung des Polynomrings, ohne dass der Benutzer manuell ein Polynom definieren muss. Zum Beispiel:

- $\deg(3) : x^3 + x + 1$: Ein Minimalpolynom dritten Grades, das in bestimmten Konstruktionen von Galois-Feldern hilfreich ist.
- $\deg(8) : x^8 + x^4 + x^3 + x + 1$: Ein Minimalpolynom achten Grades, mit dem im Körper $\mathbb{F}_{2^8}$ gerechnet werden kann. Dieser wird z.B. für den Advanced Encryption Standard AES benutzt.

Der Benutzer hat auch die Möglichkeit, ein eigenes Minimalpolynom anzugeben, falls keines der vordefinierten den Anforderungen genügt. Die Auswahl erfolgt über eine Dropdown-Liste in der Benutzeroberfläche (siehe Kapitel 4).

5.2 Erweiterter euklidischer Algorithmus (`extendedGCD`)

Der erweiterte euklidische Algorithmus wird verwendet, um den grössten gemeinsamen Teiler (ggT) von zwei Zahlen zu bestimmen und zusätzlich die Koeffizienten s und t zu finden, die die Gleichung

$$g = s \cdot a + t \cdot b$$

erfüllen, wobei g der grösste gemeinsame Teiler der zwei gegebenen Zahlen a und b ist.

Diese zusätzlichen Koeffizienten (s und t) sind von besonderem Interesse in der Zahlentheorie und in vielen kryptografischen Anwendungen, insbesondere beim Finden des modularen Inversen einer Zahl zu einem gegebenen Modulus. Der Algorithmus basiert auf wiederholten Divisionen und Rücksubstitutionen, um die Koeffizienten zu bestimmen.

Die Implementierung des erweiterten euklidischen Algorithmus im Projekt erfolgt durch die Funktion `extendedGCD`, die eine iterative Methode verwendet, um den ggT sowie die entsprechenden Koeffizienten zu bestimmen.

```
1  const extendedGCD = (a, b, field) => {
2      if(field.numberEquals(b, field.padOperator(0))) return [a, field.
           padOperator(1), b]
3      if(field.numberEquals(a, field.padOperator(0))) return [b, field.
           padOperator(1), a]
4
5      let x = b, y = a;
6      let q = field.divRest(x, y)[0];
7      let r = field.sub(x, field.mul(y, q)[0])[0]
8      let [u, s, v, t] = [field.padOperator(1), field.padOperator(0), field
           .padOperator(0), field.padOperator(1)]
9
10     while (!field.numberEquals(r, field.padOperator(0))) {
11         const newT = field.sub(v, (field.mul(q, t)[0]))[0];
12         const newS = field.sub(u, (field.mul(q, s)[0]))[0];
13         x = y;
14         y = r;
15         q = field.divRest(x, y)[0];
16
17         r = field.sub(x, field.mul(y, q)[0])[0];
18
19         v = t;
20         u = s;
21         s = newS;
22         t = newT;
23     }
24     return [y, t, s];
25 }
```

Die Variablen `v` und `u` sind temporäre Speicher für die in tabellenbasierten Implementationen verwendeten Werte t_{-1} und s_{-1}. Zurückgegeben wird:

- y: der grösste gemeinsame Teiler (ggT) von a und b,
- t: der Koeffizient, der die Gleichung $t \cdot a + s \cdot b = \text{ggT}(a, b)$ erfüllt,
- s: der Koeffizient für b in der gleichen Gleichung.

Die Funktion `extendedGCD` nutzt dabei mehrere Methoden, die in den jeweiligen grundlegenden Körperklassen (`ExtensionFields`, `FieldFp`, etc.) implementiert sind, um die arithmetischen Operationen durchzuführen. Diese Methoden stellen sicher, dass die

Berechnungen auf den entsprechenden Körpern korrekt ausgeführt werden, wie zum Beispiel die Division mit Rest oder die Addition in endlichen Körpern.

Verwendung des erweiterten euklidischen Algorithmus

Die Verwendung des erweiterten euklidischen Algorithmus in diesem Projekt liegt in der Berechnung des modularen Inversen in endlichen Körpern. Bei der Division in einem endlichen Körper $\mathbb{F}_p$ wird das Inverse eines Elements benötigt, um eine Division durchzuführen. Dies bedeutet, dass wir den Wert b^{-1} finden müssen, sodass

$$b \cdot b^{-1} \equiv 1 \mod p$$

gilt. Dasselbe gilt für $\mathbb{F}_{p^l}$, jedoch ist die modulo Operation dort durch das Minimalpolynom definiert. Die Methode `inverse` der jeweiligen Klassen verwendet den erweiterten euklidischen Algorithmus, um diese modulare Inverse zu berechnen:

```
1  inverse(b) {
2      try {
3          let [g, x] = extendedGCD(b, this.p, this);
4          if (g !== 1) {
5              return NaN; // No inverse exists
6          }
7          return (x % this.p + this.p) % this.p;
8      } catch (error) {
9          // If gcd is not 1, or error occurs, return NaN indicating no
               inverse
10         return NaN;
11     }
12 }
```

Die Methode `inverse`-Funktion ruft die `extendedGCD`-Funktion auf, um den ggT von b und p sowie die Koeffizienten zu berechnen. Wenn der ggT gleich 1 ist, existiert eine modulare Inverse, und der Wert von x kann als die Inverse verwendet werden, wobei sicherzustellen ist, dass das Ergebnis im Bereich $[0, p-1]$ liegt.

Theoretische Grundlage und Referenz

Die theoretische Grundlage des erweiterten euklidischen Algorithmus wurde basierend auf den Informationen aus [Bae23] implementiert. In diesem Artikel wird der Algorithmus anschaulich erklärt, inklusive seiner Anwendung zur Berechnung des modularen Inversen und seiner Nützlichkeit in vielen Bereichen der Informatik und Mathematik. Der Artikel stellt die iterative Natur des erweiterten euklidischen Algorithmus dar und zeigt, wie durch Rücksubstitution die Koeffizienten s und t ermittelt werden.

5.3 Reverse Polish Notation

Die Reverse Polish Notation (RPN), auch bekannt als umgekehrte polnische Notation, ist eine Schreibweise für arithmetische Ausdrücke, bei der die Operatoren den Operanden folgen. Dadurch entfällt die Notwendigkeit von Klammern, um die Reihenfolge der Operationen zu bestimmen. Diese Eigenschaft macht RPN besonders geeignet für Computer und Taschenrechner, die Operationen in einer sequentiellen Reihenfolge ausführen. [Hic]

5.3.1 Implementierung

Die Implementierung der RPN erfolgt durch die Klasse `RPNStack`, die eine Array-basierte Struktur zur Verwaltung der Werte und Operationen bereitstellt. Die Klasse beinhaltet Methoden zur Verwaltung und Manipulation des Stacks sowie zur Durchführung von Operationen. Sie ist in sich geschlossen und unabhängig von der View. Die Methoden `push`, `pop`, `swap` und `rotate` werden von View-Schicht verwendet und an die entsprechenden Knöpfe angebunden.

```
class RPNStack {
    constructor(body) {
        this.body = body;
        this.stack = [body.zero, body.zero, body.zero, body.zero];
    }

    pushValue(value) {
        this.stack.push(value)
        this.stack.splice(0, 1)
    }

    push(element) {
        if (this.isOperation(element)) {
            if (this.isValid(this.y) && this.isValid(this.x)) {
                const [result, steps, rest] = this.field[element](this.y,
                    this.x);
                if (result === null) return [result, steps, rest];

                this.popValue(); //remove x
                this.popValue(); //remove y

                this.pushValue(this.field.copyNumber(result))

                return [result, steps, rest]
            } else alert('not enough values on the stack for the
                operation')
        } else {
            this.pushValue(element);
        }
        return [null, null, null];
    }

```

```
31      pop() {
32          this.stack.pop();
33          this.stack.unshift(this.field.zero);
34      }
35
36      popValue() {
37          this.stack.pop();
38          this.stack.unshift(this.z);
39      }
40
41      swap() {
42          const [x, y] = [this.stack.pop(), this.stack.pop()];
43          this.stack.push(x, y);
44      }
45
46      rotate() {
47          const bottom = this.stack.pop();
48          this.stack.unshift(bottom);
49      }
50  }
```

5.3.2 Funktionsweise

Der Stack speichert bis zu vier Werte, wobei neue Werte unten hinzugefügt und die ältesten oben, vom Ende entfernt werden. Die `push`-Methode verarbeitet sowohl Zahlen als auch Operatoren. Bei Zahlen wird der Wert auf den Stack gelegt. Bei Operatoren werden die obersten beiden Werte entnommen, die entsprechende Operation wird durchgeführt, und das Ergebnis wird wieder auf den Stack gelegt.

Beispiel: Addition von drei Zahlen.

1. Der Benutzer gibt die Werte `5`, `8` und `3` ein, gefolgt von den Operatoren `+` und `*`.
2. Zuerst werden `5` und `8` addiert: `5 + 8 = 13`.
3. Dann wird `13` mit `3` multipliziert: `13 * 3 = 39`.
4. Das Endergebnis, `39`, verbleibt auf dem Stack.

5.3.3 X, Y, Z, T

Die vier Einträge des Stacks repräsentieren die Werte X, Y, Z und T der Reverse Polish Notation. Sie sind durch die gleich benannten `getter` Funktionen, wie

```
1    get x() {
2     return this.getStackElement(0)
3    }
```

implementiert. Visuell steht X unten und T oben im Stack. In der Implementierung steht das Ende des Stacks mit index = 3 für das unterste Extrem `X`, der Anfang für das oberste Element `T`. So spiegelt die `push` Methode des unterliegenden Arrays das Verhalten der `push` Methode der RPN. Mit `getStackIndex` wird der index 0 von innerhalb der Methode `getStackElement` auf das Ende des Arrays abgebildet.

```
1     getStackIndex(index) {
2         return this.stack.length - 1 >= index ? this.stack.length - 1 -
              index : null;
3     }
```

Dieser Ansatz wurde ursprünglich so gewählt, da der Stack zuerst mit potenziell unendlich Elementen angedacht war, wodurch die verwendung von 0 bis 3 als Indexe einfacher war als `stack.length - n`.

5.3.4 Methoden und Invariante

Die Invariante des Stacks ist seine Anzahl enthaltener Elemente. Der Stack (Variable `stack`) muss jederzeit genau vier Elemente enthalten. Initialisiert wird er im Konstruktor durch Nulleinträge. Jede Methode fügt danach gleich viele Elemente hinzu, wie sie entfernt oder umgekehrt. Dadurch verändert sich die Anzahl Elemente auf dem Stack nicht. Elemente werden von einer Benutzersicht von ausserhalb der Klasse nur durch die `push` und `pop` Methoden hinzugefügt oder entfernt. Intern werden jedoch durch viele Methoden Elemente dem Stack hinzugefügt oder entfernt um diesen neu anzuordnen.

Hilfsmethoden

Die Methode `pushValue` wird intern verwendet. Sie fügt dem Stack am Ende ein Wert hinzu und entfernt das erste Element.

Die Methode `popValue` wird ebenfalls intern verwendet. Das erste Element auf dem Stack wird entfernt, dafür wird das letzte Element, das nach dem Entfernen `z` (mit index = 2) ist, dupliziert.

Somit sind wieder vier Elemente auf dem Stack präsent.

Pop

In der `pop` Methode wird ein Wert vom Stack Ende entfernt und dafür unmittelbar mit einem Nulleintrag ersetzt.

Push

Die Methode `push`, fügt dem Stack entweder einen Operanden oder eine Operation hinzu. Handelt es sich um eine Operation wird überprüft, ob die Elemente X und Y gültige Operanden sind, ob das Resultat der angewandten Operation gültig ist. Ist beides der Fall werden X und Y mit der Methode `popValue` vom Stack entfernt und das Resultat

mit `pushValue` dem Stack hizugefügt. Handelt es sich um einen Operanden, wird dieser mit `pushValue` dem Stack hinzugefügt.

Swap

Durch die `swap` Methode, werden die Werte von `X` und `Y` vertauscht. In der Implementierung werden beide vom Stack entfernt und in umgekehrter Reihenfolge wieder hinzugefügt.

Rotate

Mit der `rotate` Methode wird der ganze Stack nach unten rotiert. Dazu wird das unterste Element (`X`) vom Ende des Stacks entfernt und am Anfang (oben) wieder hinzugefügt.

Kapitel 6

Erkenntnisse und Reflexion

In dieser Arbeit wurde die Implementierung der Arithmetik in verschiedenen mathematischen Strukturen wie Ringen und Körpern erfolgreich abgeschlossen. Während des Projekts wurden zahlreiche Herausforderungen bewältigt und wertvolle Erfahrungen gesammelt, die im Folgenden zusammengefasst werden.

Domänenwissen als Schlüssel zum Erfolg

Der Einstieg in die Problemdomäne war eine der grössten Herausforderungen zu Beginn des Projekts. Insbesondere die Konzepte der endlichen Körper, Polynomringe und Körpererweiterungen erforderten eine intensive Einarbeitung. Durch gezielte Recherche und schrittweises Lernen konnte das benötigte Wissen aufgebaut und auf die technische Umsetzung angewandt werden. Dieses Verständnis bildete die Grundlage für die Entwicklung einer flexiblen und modularen Architektur.

Iterative Entwicklung als Arbeitsmethode

Die iterative Vorgehensweise erwies sich als besonders effektiv. Zu Beginn wurde ein einfacher Rechner entwickelt, der als Prototyp diente. Dieser wurde schrittweise erweitert, um zusätzliche Anforderungen wie die Unterstützung von Polynomringen und Körpererweiterungen zu erfüllen. Durch das kontinuierliche Testen und Anpassen des Codes konnten Fehler frühzeitig erkannt und behoben werden.

Abstraktion und Code-Architektur

Ein zentrales Ziel war es, die Unterschiede zwischen Ringen (z.B. $\mathbb{Z}$) und Körpern (z.B. $\mathbb{F}_p$, $\mathbb{Q}$) sauber in der Code-Struktur abzubilden. Hierfür wurde eine Basisklasse geschaffen, die die grundlegenden Operationen bereitstellt, während die speziellen Eigenschaften der jeweiligen Zahlbereiche in den abgeleiteten Klassen implementiert wurden. Diese Abstraktion erleichtert die Erweiterbarkeit und Wartbarkeit der Anwendung.

Reverse Polish Notation (RPN) für die Benutzerfreundlichkeit

Die Entscheidung, den Taschenrechner mit der Reverse Polish Notation (RPN) zu implementieren, stellte sich als vorteilhaft heraus. RPN ermöglicht eine klare und strukturierte Eingabe komplexer Ausdrücke. Die zusätzliche Visualisierung des Stacks sowie die Implementierung von Operationen wie *Swap* und *Rotate* verbesserten die Benutzererfahrung und machten die Bedienung intuitiv.

Zusammenarbeit und Feedback

Ein wesentlicher Erfolgsfaktor war die enge Zusammenarbeit im Team sowie das Einholen von Feedback durch den Auftraggeber und Betreuer. Die Rückmeldungen führten zu wertvollen Anpassungen in der Implementierung und verbesserten die Qualität der Dokumentation. Insbesondere die Rückmeldungen zur klaren Unterscheidung von mathematischen Konzepten und deren technischer Umsetzung waren hilfreich.

Fazit

Die entwickelte Lösung zeigt, dass es möglich ist, komplexe mathematische Strukturen effizient in einer modernen Web-Anwendung umzusetzen. Die Kombination aus fundiertem Domänenwissen, einer flexiblen Architektur und benutzerfreundlichen Oberflächen ermöglicht es, die Anwendung sowohl für akademische als auch für praktische Anwendungsfälle einzusetzen.

Die iterative Arbeitsweise sowie die kontinuierliche Reflexion des Projektfortschritts haben dazu beigetragen, technische und konzeptionelle Herausforderungen erfolgreich zu bewältigen. Das Projekt bietet zudem eine solide Grundlage für zukünftige Erweiterungen und Weiterentwicklungen.

Literaturverzeichnis

[Bae23] Baeldung. Extended euclidean algorithm. https://www.baeldung.com/cs/extended-euclidean-algorithm, 2023.

[Hic] David G. Hicks. Rpn. https://www.hpmuseum.org/rpn.htm.

[Mül23] Andreas Müller. *Lineare Algebra: Eine anwendungsorientierte Einführung.* Springer Berlin, Heidelberg, 2023.