

Fakultät für Mathematik und Informatik
Bachelor of Science Informatik

Bachelorarbeit

Ein Computeralgebrasystem in Rust



Verfasser	Bernd Haßfurthner <nerglom@posteo.de>
Matrikel-Nr.	4372280
Betreuerin	Prof. Dr. Lena Oden
Datum	7. Mai 2022

Inhaltsverzeichnis

1	Einleitung	3
2	Vorstellung der Programmiersprache Rust	4
2.1	Einführendes Beispiel Ownership und Borrowing	4
2.2	Was ist Rust und warum verwenden?	5
2.3	Grundlegendes	6
2.4	Enums	8
2.5	Generics	8
2.6	Operatorenüberladung	10
2.7	Ownership, Borrowing und Lifetimes	11
3	Implementierung des CAS	14
3.1	Grundregeln und Annahmen	14
3.2	Funktionsumfang	14
3.3	Überlauf und Ungenauigkeit	15
3.4	Grundlegende Datenstruktur	17
3.5	Grundtypen des CAS	18
3.6	Datenstruktur in Rust	19
4	Parser für mathematische Ausdrücke	24
4.1	Prüfung von bestehenden Bibliotheken zum Parsen	24
4.2	Implementierung der lexikalischen Analyse bzw. Tokenizer	25
4.3	Implementierung Parser	25
5	Grundfunktionalitäten	27
5.1	Änderung des Terms mithilfe Operatorenüberladung	27
5.2	Addieren und Multiplizieren (Standardauswertung von Termen)	28
5.3	Umgang mit Brüchen, Potenzen und Wurzeln	30
5.4	Erweitbarkeit in Rust und Beispiele	30
5.5	Implementierung von Substitution	31
5.6	Implementierung von expand	33
5.7	Implementierung von simplify	35
6	Erweiterte Funktionalitäten (Beispiele)	36
6.1	Terme mit Fließkommazahlen berechnen oder auch Ungenauigkeit in Kauf nehmen	36
6.2	Implementierung der limit-Funktion	36
6.3	Implementierung einer to string-Methode	36
7	Vergleich zu SymPy	37
7.1	Korrektheit und Unterschiede gegenüber SymPy	37
7.2	Performancevergleich	37
8	Fazit	38
8.1	Erweiterungs- und Verbesserungsideen des Systems	38
8.2	Vor- und Nachteile Rust	38

1 Einleitung

Computeralgebrasysteme - im folgender Arbeit abgekürzt CAS - nehmen in verschiedenen Bereichen der Wissenschaft eine wichtige Rolle ein. Sie ermöglichen es einerseits mathematische Ausdrücke mit Variablen darzustellen, diese Terme zu verändern und mit diesen zu arbeiten. Andererseits lassen sich wesentlich genauere Berechnung durchführen, indem z.B. nicht alle Ausdrücke in einem Term ausgewertet werden, wenn es hierdurch zu Ungenauigkeiten kommen kann.

Beispiele von Verwendungen eines CAS

In der vorliegenden Arbeit soll ein solches CAS in Ansätzen mit der Programmiersprache Rust entwickelt werden. Es soll untersucht werden, inwieweit das Ownership- und Borrowing-System von Rust bei einem solchen komplexen System bei der Entwicklung unterstützt und wie die Performance zur Laufzeit ausfällt. Da System soll dabei einfach erweiterbar sein. Dabei sollen sowohl neue mathematische Funktionen einfach integriert werden können, als auch die Terme mit neuen Regeln zusammenfassbar sein.

Einen besonderen Stellenwert nimmt SymPy ein, da dieses System einerseits als Open-Source-Projekt verfügbar ist, als auch der Code in Python geschrieben ist und leicht verständlich ist. Der Code soll dabei natürlich nicht 1-zu-1 in Rust neu geschrieben werden, sondern mit den Möglichkeiten von Rust erweitert werden, auch wenn natürlich nur ein Teil des Umfangs von SymPy in dieser Arbeit möglich sein wird.

2 Vorstellung der Programmiersprache Rust

2.1 Einführendes Beispiel Ownership und Borrowing

Bevor Rust vorgestellt wird und die genauen Regeln des Ownership und Borrowing erläutert werden, soll das Prinzip anhand eines Beispiels verdeutlicht werden. Hierzu wird ein `struct` - vergleichbar mit Klassen - angelegt, welches z.B. eine große Matrix beinhaltet, die verarbeitet werden soll.

Codebeispiel 1: Einführendes Beispiel struct

```
struct LargeMatrix {  
    matrix: Vec<i32>,  
}
```

Im ersten Beispiel wird eine Funktion erstellt, die das Ownership der Matrix übernimmt und nach der Verarbeitung den Speicherplatz freigibt. Dies geschieht automatisch, da der Gültigkeitsbereich der Variable `matrix_fn` verlassen wird. In diesem Kontext bedeutet Ownership, dass der Wert vom Hauptprogramm in die Funktion verschoben wird und die neue Variable `matrix_fn` das alleinige Recht besitzt diesen Wert zu nutzen. Das Programm lässt sich so nicht kompilieren. Die zweite Ausgabe schlägt dabei fehl, da beim Aufruf der Funktion der Wert in die Funktion verschoben und freigegeben wird, der Zugriff auf den entsprechenden Speicherbereich im Hauptprogramm wäre undefiniert.

Codebeispiel 2: Einführendes Beispiel Ownership abgeben

```
fn take_ownership(matrix_fn: LargeMatrix) {  
    // ...  
}  
  
let matrix = LargeMatrix { matrix: vec![] };  
println!("{:?}", matrix);  
take_ownership(matrix);  
println!("{:?}", matrix); // <- Fehler
```

Die zweite Funktion gibt das Ownership der Matrix wieder zurück, der Wert wird also wieder zurückgeschoben und kann nach dem Funktionsaufruf weiter verwendet werden. Dies geschieht aber nur, wenn der Variablen im Hauptprogramm der Rückgabewert zugewiesen wird. Hierzu muss die Variable `matrix` explizit als veränderbar mit dem Schlüsselwort `mut` definiert werden. Ein Nachteil dieser Methode ist, dass nicht klar ist, ob es sich um die originale Matrix handelt, oder über eine neu initialisierte Matrix.

Codebeispiel 3: Einführendes Beispiel Ownership zurückgeben

```
fn give_back_ownership(matrix_fn: LargeMatrix) -> LargeMatrix {  
    // ...  
    // Originale Matrix zurückgeben  
    matrix_fn  
    // Oder neue Matrix zurückgeben  
    // LargeMatrix { matrix: vec![] }  
}  
  
let mut matrix = LargeMatrix { matrix: vec![] };  
println!("{:?}", matrix);
```

```
matrix = give_back_ownership(matrix);  
println!("{:?}", matrix);
```

Im nächsten Beispiel erhält die Funktion nur eine Referenz auf die Matrix, gekennzeichnet durch das `&`-Zeichen. In Rust wird dieses Prinzip als “Borrowing” bezeichnet (quelle: <https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html>). Wie bereits im ersten Beispiel wird beim verlassen der Funktion der Speicherplatz der Variablen freigegeben, diesmal handelt es sich aber nur um den Zeiger und nicht um die eigentlichen Daten. Eine Verwendung nach dem Funktionsaufruf ist daher weiterhin möglich.

Da Variablen in Rust standardmäßig unveränderbar sind, ist ebenso sichergestellt, dass die Matrix in der Funktion nicht verändert worden ist.

Codebeispiel 4: Einführendes Beispiel Unveränderliche Referenz

```
fn take_reference(matrix_ref: &LargeMatrix) {  
    // ...  
}  
  
let matrix = LargeMatrix { matrix: vec![] };  
println!("{:?}", matrix);  
take_reference(&matrix);  
println!("{:?}", matrix);
```

Im letzten Beispiel soll nun durch die Verwendung des `mut`-Schlüsselworts ermöglicht werden, die Daten der originalen Matrix zu verändern. Ansonsten gibt es keine Änderung zum vorherigen Beispiel, beim verlassen der Funktion wird auch wieder nur der Zeiger freigegeben.

Codebeispiel 5: Einführendes Beispiel Veränderliche Referenz

```
fn take_mutable_reference(matrix_ref: &mut LargeMatrix) {  
    matrix_ref.matrix[0] = 0;  
    // ...  
}  
  
let mut matrix = LargeMatrix { matrix: vec![] };  
println!("{:?}", matrix);  
take_mutable_reference(&mut matrix);  
println!("{:?}", matrix);
```

Natürlich können die beiden letzten Funktionen auch einen Rückgabewert besitzen, der mit dem der eigentlich Matrix nichts zu tun hat. Eine besondere Stärke sind die Regeln, die für das Borrowing gelten. So ist z.B. zu jeder Zeit nur eine veränderbare Referenz auf eine Variable erlaubt, es ist daher leicht nachvollziehbar welcher Programmabschnitt die Daten tatsächlich verändern kann. Race-Conditions in nebenläufigen Anwendungen sind somit ausgeschlossen. Rust bietet für solche Einsatzzwecke spezielle Datenstrukturen wie (siehe Seminararbeit und Beispiel). Die genauen Regeln zu Ownership und Borrowing werden noch vorgestellt.

2.2 Was ist Rust und warum verwenden?

Rust ist eine statisch typisierte Programmiersprache, welche vor allem mit den Eigenschaften Performance, Verlässlichkeit und Produktivität eine Alternative zu anderen Sprachen anbieten kann. Version 1.0 wurde 2015 veröffentlicht, seitdem gibt es mit der “Rust 2018 Edition” und “Rust 2021 Edition” eine stetige Weiterentwicklung.

Die Performance begründet sich durch zwei wesentliche Aspekte. So werden alle Regeln die das Ownership und das Borrowing betreffen bereits zur Compilezeit durchgeführt. Ein zusätzlicher Overhead zur Laufzeit des Programms wird so vermieden. Der andere Aspekt ist die Implementierung von Featur in Rust. So wird beispielsweise Monomorphisierung zur Compilezeit von Generics eingesetzt. Für jeden implementierten Typen des Generics gibt es somit eine eigene Implementierung im Compilat. Hierdurch entfallen Typechecks wie sie z.B. in Java möglich sein können.

Die Verlässlichkeit beruht ebenfalls auf dem Ownership- und Borrowing-System. Hierdurch ist zu jeder Zeit klar, welche Variablen welche Daten enthalten. Ein unbeabsichtigtes Verändern der Daten oder Dangling References werden ebenfalls bereits zur Compilezeit ausgeschlossen, dieses Prinzip funktioniert ebenfalls in nebenläufigen Anwendungen. Zudem stellt Rust sicher, dass man nicht außerhalb eines Puffers lesen oder schreiben kann, dies kann aber zu Umständen zu einem Programmabbruch führen. Das spezielle `Option enum` bietet eine fehlerunanfällige Variante des `NULL`-Werts, das in Rust nicht existiert.

Um die Produktivität zu unterstützen bieten die beiden Tools `rustup` und `cargo` alles notwendige um den kompletten Entwicklungsprozess eines Rust-Programms zu erfüllen. Dies betrifft z.B. konkret eine Abhängigkeitsverwaltung, die Erstellung von Dokumentation und das ausführen von Tests und Benchmarks.

Wo nötig setzt Rust auf etablierte Lösungen, so wird LLVM genutzt um ausführbare Programme zu erzeugen. Rust wird mittlerweile in Verschiedenen Projekten genutzt (z.B. Android OS, Linux Kernel, Windows-API). Alles zusammen rechtfertigt zumindest die Überlegung ein komplexes System - wie eben ein CAS - in Rust zu implementieren.

2.3 Grundlegendes

Expression und Statements Rust unterscheidet bei Code-Anweisungen zwischen Statements und Expressions. Statements führen lediglich einen Codeblock aus, während Expressions stets einen Rückgabewert besitzen. Nahezu alle Sprachelemente in Rust sind als Expression implementiert.

Codebeispiel 6: Expression Einführendes Beispiel

```
let test_var = if bedingung_1 { false } else { true };
```

struct Um eigene Datentypen zu definieren, werden `structs` verwendet. Die Deklaration eines `struct` gibt nur Felder und Feldtypen an. Methoden und Funktionen werden erst nach der Deklaration hinzugefügt, dabei sind mehrere Implementierungsblöcke möglich. Die Implementierungen können Methoden inkl. einer Referenz auf die aktuelle Instanz oder Funktionen ohne entsprechende Referenz enthalten.

Dabei sollen `structs` den Vorteil der Übersichtlichkeit und dieselbe Syntax wie `enums` bieten (vgl. [1, S. 91]). Klassen gibt es in Rust nicht.

Das folgende Beispiel deklariert ein `struct` mit einem Feld. Anschließend wird ein Implementierungsblock angegeben, der jeweils eine Funktion und eine Methode enthält.

Codebeispiel 7: struct

```
struct MyStruct {
    my_field: i32,
}

impl MyStruct {
    fn new() -> MyStruct {
```

```
    MyStruct {
        my_field: 5,
    }
}

fn do_something(&self) {}
}
```

trait In Rust gibt es weder Vererbung durch **structs**, noch klassische Interfaces. Mit **traits** kann aber ein ähnliches Verhalten nachgebildet werden. Dabei findet bei der Definition des **trait** die Deklaration von Methoden und Funktionen statt. Anschließend können **structs** und **enums** ein oder mehrere **traits** implementieren. Der Traitname wird schlussendlich z.B. in der Parameterdeklaration von Funktionen verwendet.

Für das bereits bestehende **struct** soll ein **trait** erstellt und implementiert werden.

Codebeispiel 8: trait

```
trait MyTrait {
    fn trait_method(&mut self) -> bool;
}

impl MyTrait for MyStruct {
    fn trait_method(&mut self) -> bool {
        true
    }
}
```

match Als Alternative zur **switch**-Anweisung bietet Rust **match**. Im Gegensatz zu anderen Sprachen muss bei **match** jeder mögliche Wert einer Variablen geprüft werden. Durch einen Variablennamen als letzter zu prüfender Wert wird dieser verallgemeinert. Alternativ kann der Wert unberücksichtigt gelassen werden.

Bei der Auswertung der Variablen ist es zudem erlaubt mehrere Werte gleichzeitig anzugeben. Da **match** ebenfalls als Expression umgesetzt ist, kann ein Rückgabewert wieder einer Variablen zugeordnet werden. Je nach Wert können verschiedene Aktionen ausgeführt werden, die ursprüngliche Variable geändert werden und ebenfalls ein Rückgabewert definiert werden.

Codebeispiel 9: match

```
let mut var = 19;

let match = match var {
    1 => {
        var = 5;
        String::from("klein")
    }
    2..=10 => String::from("bis 10"),
    11 | 13 | 17 => String::from("Primzahl"),
    i => i.to_string(),
    // _ => String::from("Nichts"),
}
```

```
};
```

2.4 Enums

Der Aufzählungstyp `enum` spielt in Rust eine wichtige Rolle. So wird z.B. mit dem `enum Result` das Error-Handling in Rust ermöglicht. Weiterhin gibt es mit dem `Option` `enum` wie bereits erwähnt eine Alternative um fehlende `NULL`-Werte in Rust zu darzustellen.

Jede Ausprägung eines `enum` kann verschiedene Werte enthalten. Wie bereits `structs` können Implementierungsblöcke definiert und Traits implementiert werden. Mit dem bereits vorgestellten `match` kann geprüft werden, welche Ausprägung das `enum` hat. Alternativ funktioniert dies auch mit der `if let`-Anweisung, die es erlaubt nur auf eine Ausprägung zu überprüfen.

In diesem Beispiel wird ein `enum` mit verschiedenen Werten definiert, die dann über eine `match`-Anweisung konkret abgefragt werden können. Zudem gibt es wieder einen Implementierungsblock für Funktionen und Methoden und es wird ein `trait` implementiert. Die Syntax ist dabei identisch wie bereits bei `structs`.

Codebeispiel 10: enum

```
enum MyEnum {
    Entry1 { c: i32, m: i32, y: i32, k: i32 }, // Anonymes struct
    Entry2(i32, i32, i32), // 3 unbenannte Werte
    Entry3, // ohne Daten
}

let x = MyEnum::Entry2(42, 42, 42);
match x {
    MyEnum::Entry2(v1, v2, v3) => { ... },
    _ => { ... }
};

impl MyEnum { ... }
impl MyTrait for MyEnum { ... }
```

2.5 Generics

Rust bietet ebenfalls generische Typparameter an. Diese können bei `struct`, `enum`, `trait` und Funktionen verwendet werden. Um eine hohe Performance zu gewährleisten, wird Monomorphisierung zur Compilezeit eingesetzt (vgl. [7, S. 196 ff.]). Dies bedeutet, dass z.B. eine entsprechende Funktion speziell für die verwendeten Typen implementiert wird.

Dies wird exemplarisch für das bestehende `Option` `enum` veranschaulicht, welches im fertigen Programm zwei mal mit unterschiedlichen Typen existiert.

Codebeispiel 11: Monomorphisierung Veranschaulichung [2]

```
// enum im Quellcode
enum Option<T> {
    Some(T),
    None,
}
```



```

let integer = Some(5);
let float = Some(5.0);

// enum nach Monomorphisierung
enum Option_i32 {
    Some(i32), None,
}
enum Option_f64 {
    Some(f64), None,
}
fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}

```

Im Zusammenspiel mit Traits kann ein entsprechender Typparameter begrenzt werden. Ein Nachteil der Monomorphie ist, dass ein Typparameter immer nur von einem konkreten Typ ersetzt werden kann. Eine Lösung hierfür bieten **trait objects**.

Im folgenden Beispiel dazu werden zuallerst ein **trait** und zwei **structs** definiert, die das **trait** implementieren. Das **ScreenTraitObject** kann im Vektor **components** beide Structs unabhängig voneinander vorhalten, da **trait objects** verwendet werden. **ScreenGeneric** benutzt allerdings nur einen einfachen Generic-Parameter für den Vektor, sodass tatsächlich nur eine konkrete Implementierung von **Draw** hinzugefügt werden.

Durch den **Box**-Typ werden die Werte auf dem Heap abgelegt, da nicht von vornherein klar ist, wie viel Speicher auf dem Stack benötigt wird.

Codebeispiel 12: Trait Boundaries [4]

```

trait Draw {}

struct Button {}
struct SelectBox {}

impl Draw for Button {}
impl Draw for SelectBox {}

struct ScreenTraitObject {
    components: Vec<Box<dyn Draw>>,
}
// Bevorzugt, wenn nur wenige Typparameter und/oder Einschränkungen
struct ScreenGeneric<T: Draw> {
    components: Vec<T>,
}
// Oder falls mehrere Typparameter und/oder Einschränkungen
struct ScreenGeneric2<T, U>
where
    T: Draw + Trait2 + Trait3,
    U: Trait4
{}

ScreenTraitObject {
    components: vec![

```

```

        Box::new(Button {}),
        Box::new(SelectBox {}),
    ],
};

ScreenGeneric {
    components: vec![
        Button {},
        SelectBox {}, // Fehler, da SelectBox != Button
    ],
};

```

2.6 Operatorenüberladung

In Rust können Operatoren überladen werden und somit für eigene als auch bestehende Typen implementiert und erweitert werden. Diese Funktionilität kann im CAS dazu genutzt werden, um z.B. einen bestehenden Term ohne Funktionsaufruf zu verändern. Stattdessen können gewohnte Sprachmittel verwendet werden. Auch ist es möglich die Prüfung auf Gleichheit mit einem anderen Datentyp zu implementieren.

Im Beispiel kann das erstellte `enum` mit sich selbst addiert und mit dem primitiven Datentyp `i32` verglichen werden. Dabei werden die bereits existierenden Operatoren “+” und “==” verwendet.

Codebeispiel 13: Operatorenüberladung

```

enum Number {
    Int(i32),
    Float(f64),
}

impl std::ops::Add<Number> for Number {
    type Output = Number;

    fn add(self, rhs: Number) -> Number {
        match (self, rhs) {
            (Number::Int(v1), Number::Int(v2)) => Number::Int(v1 + v2),
            // Weitere Implementierung
        }
    }
}

let n1 = Number::Int(1);
let n2 = Number::Int(2);
let n3 = n1 + n2;

impl PartialEq<i32> for Number {
    fn eq(&self, other: &i32) -> bool {
        match self {
            Number::Int(v) => v == other,
            _ => false,
        }
    }
}

```

```

    }
}

if n3 == 3 { ... }

```

2.7 Ownership, Borrowing und Lifetimes

Im folgenden sollen die Konzepte Ownership, Borrowing und Lifetimes kurz vorgestellt werden, diese tragen wesentlich zur erwähnten Speichersicherheit bei. Der Compiler beachtet dabei die folgenden Regeln.

- Jeder Wert ist einer Variablen zugewiesen, dem **Owner**
- Jeder Wert kann nur einen **Owner** besitzen
- Verlässt der **Owner** den Gültigkeitsbereich, wird die Variable ungültig und im Normalfall der Speicherplatz freigegeben
- Zu jeder Zeit kann es eine beliebige Anzahl an nicht veränderbaren Referenzen geben oder exakt eine veränderbare Referenz
- Referenzen müssen immer gültig (d.h. deklariert sein und der Owner muss mindestens denselben Gültigkeitsbereich wie die Referenz haben) sein

Das Ownership-Prinzip deckt dabei bereits die ersten drei Regeln ab. Zu jedem Zeitpunkt kann jeder Wert nur durch eine Variable angesprochen werden. Liegt der Wert auf dem Stack oder wird das **Copy**-Trait implementiert wird bei einer Zuweisung der Wert kopiert. Liegt der Wert hingegen auf dem Heap, wie z.B. bei einem **Vector** oder einem **String** wird das Ownership an die neue Variable übergeben.

Codebeispiel 14: ownership

```

let mut a = 2;
let b = a;
a = 4;
println!("{}", a, b); // Ausgabe: 42

let s1 = String::from("hello");
let s2 = s1;
println!("{}", s2); // Ok
println!("{}", s1); // Fehler, Ownership wurde s2 übertragen

```

Um nun nicht ständig den Owner von Daten ändern zu müssen z.B. bei einem Funktionsaufruf werden mit dem Borrowing Referenzen erstellt. Hierauf beziehen sich die beiden letzten Regeln. Dabei gilt außerdem, dass es nur veränderbare Referenzen geben kann, wenn der Owner veränderbar ist und falls es eine veränderbare Referenz gibt, kann der Wert auch nur exklusiv durch diese verändert werden. Zur Referenzierung und Dereferenzierung kommen die **&**- und *****-Operatoren zum Einsatz.

Codebeispiel 15: borrowing

```

// Nicht zulässig, da zwei mutable Referenzen
let mut a = 5;
let b = &mut a;

```

```

let c = &mut a;
*b = 1;

// Nicht zulässig, da mutable und nicht mutable gemischt
let mut a = 5;
let b = &mut a;
let c = &a;
let d = &a;
*b = 1;

// Zulässig, da nur lesende Referenzen
let a = 5;
let b = &a;
let c = &a;

fn append_string(string1: &mut String, string2: &String) {
    string1.push_str(string2);
} // Nur Referenzen verlassen den Gültigkeitsbereich, nicht die Owner!

let mut s1 = String::from("Fern");
let s2 = String::from("Uni");
append_string(&mut s1, &s2);
println!("{}", s1); // Ausgabe: FernUni

```

Schlussendlich verhindern lifetimes, dass die Referenzen für das Borrowing ungültig werden, und die Variable - und somit der Wert - vor den Referenzen freigegeben wird. Die Illustrierung der Lifetimes entspricht dabei den Codeblöcken, in denen die Referenzen definiert sind. Die inneren Lifetimes im Beispiel könnten dabei aber auch Funktionsaufrufe sein. Das erste Beispiel ist ungültig, da die Lifetime des Wertes kleiner ist, als die Referenz, die darauf zeigen soll.

Das zweite Beispiel ist dagegen gültig, da der eigentliche Wert erst nach der Referenz freigegeben wird.

Codebeispiel 16: lifetime Veranschaulichung [3]

```

{
    let r; // -----+-- 'a
           //          |
    {
        let x = 5; // -+-- 'b |
        r = &x;    // |      |
    }             // -+      |
                 //          |
    println!("r: {}", r); //          |
}                  // -----+

{
    let x = 5; // -----+-- 'b
              //          |
    let r = &x; // --+-- 'a |
              // |      |
    println!("r: {}", r); // |      |

```

```

}                                     // --+      |
                                     // -----+

```

In den meisten Fällen kann der Compiler die lifetimes von Variablen und Referenzen selbst bestimmen. Eine explizite Deklaration der Lifetimes ist nötig, wenn beispielsweise eine Funktion mehr als eine Referenz entgegennimmt und auch wieder zurückgibt. Dies soll an einem Beispiel verdeutlicht werden. Die Lifetimedeklaration `'a` gibt an, dass sowohl die beiden Auswahlmöglichkeiten als auch der Rückgabewert dieselbe Lifetime besitzen müssen. Während dies bei der ersten Zuweisung erfüllt ist, wird eine Auswahlmöglichkeit bei der zweiten Zuweisung erst in einer inneren Lifetime deklariert, dies stimmt dann aber entsprechend nicht mit dem Rückgabewert überein.

Codebeispiel 17: lifetime Beispiele [5]

```

fn main() {
    // Funktioniert, da Werte dieselbe lifetime wie Rückgabewert haben
    let mult1 = 1;
    let mult2 = 2;
    let multiplikator = {
        let zaehler: i32 = 3;
        multiplikator_waehlen(&mult1, &mult2, &zaehler)
    };

    // Funktioniert nicht, da mult2 vor multiplikator freigegeben wird
    let mult1 = 1;
    let multiplikator = {
        let mult2 = 2;
        let zaehler: i32 = 3;
        multiplikator_waehlen(&mult1, &mult2, &zaehler)
    };
}

fn multiplikator_waehlen<'a, 'b>(
    mult1: &'a i32,
    mult2: &'a i32,
    decision_maker: &'b i32,
) -> &'a i32 {
    if *decision_maker < 5 {
        mult1
    } else {
        mult2
    }
}

```

3 Implementierung des CAS

3.1 Grundregeln und Annahmen

Bevor überlegt wird, wie der Aufbau des CAS aussieht und welche Typen benötigt werden, müssen noch ein paar Vorüberlegungen gemacht werden, welche Grundregeln und Annahmen im System gelten sollen. Es sollen folgende Zahlenräume betrachtet werden:

- \mathbb{N} die Menge der natürlichen Zahlen mit 0, $\{0, 1, 2, 3, \dots\}$
- \mathbb{Z} die Menge der ganzen Zahlen, $\{\dots - 3, -2, -1, 0, 1, 2, 3, \dots\}$
- \mathbb{Q} die Menge der rationalen Zahlen, $\{\dots - \frac{2}{1}, -\frac{1}{2}, -\frac{1}{1}0, \frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \dots\}$
- \mathbb{R} die Menge der reellen Zahlen, z.B. $\sqrt{2}$ oder π

Der komplexe Zahlenbereich \mathbb{C} wird im Rahmen dieser Arbeit nicht betrachtet.

Um insgesamt weniger Bedingungen für mathematische Operationen zu haben, ist geplant folgende Vereinfachungen und Bedingungen anzunehmen. so müssen alle Implementierungen nur die Operationen Addieren und Multiplizieren berücksichtigen.

- Jede Subtraktion ist im Grunde eine Addition. Für Zahlen bedeutet dies $5 - 3 = 5 + (-3)$ und für Symbole bzw. Terme anderer Art $a - b = a + (-1 * b)$. Eine Besonderheit sollen Terme der Art $1 - (a + b + 1)$ darstellen, diese sollen zu $1 + (-1 * a) + (-1 * b) + (-1)$ umgeformt werden und nicht zu $1 + (-1 * (a + b + 1))$
- Bei Divisionen werden Zahlen zu rationalen Zahlen, also $2/3 = \frac{2}{3}$ und $-(2/3) = \frac{-2}{3}$. Symbole und Terme werden zu einer Multiplikation vereinfacht, also $a/b = b^{(-1)} * a$ und $(a * b * c)/(d + 1) = (d + 1)^{-1} * (a * b * c)$
- Das Kommutativgesetz soll beachtet werden: $x + y = y + x$ also auch $2 * (x + y) + (y + x) * 2 = 4 * (x + y)$
- Das Assoziativgesetz soll beachtet werden: $a + (b + c) = (a + b) + c = a + b + c$
- Das Distributivgesetz soll beachtet werden: $a * (x + y) = a * x + a * y$

3.2 Funktionsumfang

Da wie bereits in der Einleitung gezeigt ein CAS sowohl einen sehr breiten Umfang als auch sehr spezialisiert sein kann, soll noch definiert werden, welche Operationen erlaubt sein sollen und welche Art von Termen verarbeitet werden sollen können:

- Addition von Zahlen (Ganzzahlen, Fließkommazahlen, rationalen Zahlen), z.B. $1 + 2.1 = 3.1$
- Addition von Symbolen und Termen, z.B. $x + y = x + y$ oder $x + (2 * y) + 2 + (2 * y) + x = 2 * x + 4 * y + 2$
- Multiplikation von Zahlen (Ganzzahlen, Fließkommazahlen, rationalen Zahlen), z.B. $1 * 2.1 = 2.1$
- Multiplikation von Symbolen und Termen, z.B. $x * y * x = x^2 * y$
- Auswertung von Potenzen von Zahlen, Symbolen und Terme inkl. Potenzregeln in Hinblick auf Genauigkeit des Ergebnis, z.B. $x^a * x^b = x^{(a+b)}$ oder $x^a * y^a = (x * y)^a$

- Auswertung von Funktionen, die dynamisch hinzugefügt werden können sollen. Funktionen die dem CAS nicht bekannt sind, tauchen weiterhin in Termen auf, werden aber nicht ausgewertet, z.B. $\sin(x)$ oder \sqrt{x}
- Auswertung von Konstanten falls bekannt und gewollt
- Die Ausführungsreihenfolge in Termen lautet: Konstanten und Variablen \rightarrow Funktion \rightarrow Potenz \rightarrow Multiplikation \rightarrow Addition

Darüberhinaus sollen Terme soweit wie möglich mit bereits bestehenden Sprachmitteln erweitert und verglichen werden können, hierzu werden einige Operatorüberladungen implementiert. Bei der Auswertung der Terme soll die Möglichkeit bestehen auch ungenaue numerische Ergebnisse zu berechnen. Funktionen und Konstanten sollen einfach hinzugefügt werden können.

3.3 Überlauf und Ungenauigkeit

Ein bereits zu Beginn prominentes Problem für ein CAS ist das Handling von Ganzzahl-Überläufen und die Ungenauigkeiten von Rechnung mit Fließkommazahlen (siehe IEEE754). So gibt das folgende Programm in Rust einen falschen Wert für die Addition aus und bricht anschließend ab.

Codebeispiel 18: Überlauf und Ungenauigkeit

```
let f1 = 0.1;
let f2 = 0.2;
println!("{}", f1 + f2);

let mut x = i32::MAX;
x += 1;
println!("{}", x);

// Ausgabe
0.300000000000000004
thread main panicked...
```

Ein Blick zu SymPy zeigt, dass bereits hier Lösungen direkt aus Python verwendet werden können, die in Rust nicht existieren. So besitzen Ganzzahltypen bereits eine unbegrenzte Präzision ([6]). Für Fließkommazahlen ist die Klasse `Float` implementiert, die eine unbegrenzte Genauigkeit erlaubt. Dabei kann mit dem zweiten Parameter die eigentliche Präzision bestimmt werden. Es wird darüberhinaus angeraten den eigentlichen Wert als String zu übergeben, da der native float-Typ in Python bereits eine Ungenauigkeit hervorrufen kann. (siehe `src code class Float(Number):`)

Codebeispiel 19: SymPy Precision

```
x = fakultaet(100) // 158 Stellen
print(type(x))
// <class int>

print(sympy.Float('0.1', 100) + sympy.Float('0.2', 100))
// 0.3000000000000000000000000000000000000000000000000000000...

print(float(0.1) + float(0.2))
// 0.30000000000000004
```

In Buch und Buch (Quelle) ist eine allgemeine Implementierung für solche Werte angegeben. Hiervon könnten wir in dieser Arbeit natürlich gebrauch machen und diese Typen selbst implementieren, da Rust diese Typen leider nicht selbst anbietet. Allerdings existieren bereits die beiden Crates (Abhängigkeiten) `num` (link) und `bigdecimal` (link, baut auf `num` auf), die Typen mit unbegrenzter Präzision anbieten.

Folgend wird die Performance zwischen den primitiven Typen auf dem Stack, den primitiven Typen auf dem Heap und den Crates verglichen. Wie zu erwarten ist die Speicherung auf dem Stack am effizientesten, der Zugriff auf dem Heap kostet Zeit. Der zusätzliche Zeitverlust bei den Crates lässt sich zum einen dadurch erklären, dass immer ein Überlauf geprüft werden muss um den Wert richtig zu verarbeiten, andererseits arbeitet der Typ `BigInt` mit einem Vektor. Wie schon erwähnt wird hier bei einem Zugriff zusätzlich geprüft, ob der Index gültig ist, was ebenfalls Zeit kostet.

```
running 6 tests
test loop_decimal    ... bench:   7,447,000 ns/iter (+/- 474,681)
test loop_float      ... bench:    36,975 ns/iter (+/- 809)
test loop_float_heap ... bench:    74,324 ns/iter (+/- 920)
test loop_int        ... bench:    18,505 ns/iter (+/- 396)
test loop_int_heap   ... bench:    50,293 ns/iter (+/- 1,683)
test loop_num        ... bench:   548,640 ns/iter (+/- 42,989)
```

Leider ist es mit Rust-Benchmarks noch nicht möglich den Speicherverbrauch zu messen. Allerdings können wir diesen für unsere Beispiele auch selbst gut bestimmen. Hierbei geht es nur um die eigentlichen Werte, mit denen Berechnungen durchgeführt werden.

Primitive Typen Stack Der Datentyp `i64` und `f64` verbrauchen 8 Bytes. Es werden also nie mehr als $8 \cdot 4 = 32$ Bytes für die eigentlichen Daten benötigt. Die Variable `x`, die das Ergebnis enthält beansprucht davon 8 Bytes. Der Schleifenparameter `i` wird ebenfalls als `i64` (bzw. `i32` im Falle der Benchmarks mit `f64`) behandelt durch die Addition mit `x` (bzw. dem TypeCasting) und verbraucht ebenfalls 8 Bytes. Bei der Addition `x+i` wird nun wie bereits erwähnt bei einem Funktionsaufruf eine Kopie der Werte erstellt, die den `Copy-Trait` implementiert haben. Der Typ `i64` und `f64` haben diesen implementiert (link src i64 impl copy) und somit wird sowohl von `x` als auch von `i` zur Addition eine Kopie erzeugt.

Primitive Typen Heap Der Speicherverbrauch der eigentlich Werte ist wie bei den primitiven Typen auf dem Stack. Davon entfallen aber 8 Bytes auf dem Heap statt auf dem Stack. Zusätzlich befindet sich ein Zeiger auf dem Stack durch `Box`. Die Größe hiervon hängt von der verwendeten Architektur ab.

num und bigdecimal Bei dem Datentyp `BigInt` wird einerseits ein Vorzeichen (enum) verwendet, welches 1 Byte verbraucht (<https://fasterthanli.me/articles/peeking-inside-a-rust-enum>) als auch der Datentyp `BigUint` der wiederum einen Vektor mit `BigDigit` erhält. Schaut man sich diese Definition an, kann dieser Typ zwischen 4 Bytes (`u32`) und 16 Bytes (`i128`) verbrauchen. Während des Debuggens auf verschiedenen Systemen wird in diesem Beispiel der Typ `u64` angenommen und somit ein Speicherplatz von 8 Bytes benötigt. Mit größer werdenden Zahlen steigt allerdings dieser Verbrauch, da sich der Vektor vergrößert. Auch liegen nicht mehr alle Daten auf dem Stack, sondern alle `BigDigits` auf dem Heap, was die Zugriffsgeschwindigkeit wie bereits gezeigt erhöht.

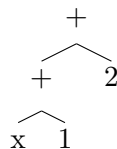
Der Typ `bigdecimal` verwendet laut Beschreibung den Typ `BigInt` und zusätzlich einen 64-Bit-Integer um die Position der Dezimalstelle zu bestimmen. Hier erhöht sich der Speicherverbrauch also nur unwesentlich.

Entscheidung der verwendeten Typen Im Rahmen dieser Arbeit ist zumindest eine eigene Implementierung der Typen für unbegrenzte Präzision nicht das Ziel. Genausogut können auch die bereits bestehenden Crates verwendet werden. Da dadurch aber die Performance leidet, wenn auch wahrscheinlich nicht relevant für spätere Beispiele und Tests, ist wiederum die Implementierung mit primitiven Typen von Vorteil.

Da Rust Generics unterstützt wurde die Entscheidung getroffen, das CAS in Teilen mit Generics umzusetzen, sodass beide Kombinationen verwendet werden können. Dabei sollen alle Funktionen des Systems mit den primitiven Typen funktionieren und einige Beispielimplementierungen mit den Crate-Typen gemacht werden. Dies erlaubt einerseits eine spätere Erweiterung der Crate-Typen als auch die Entwicklung der eigenen Typen. Darüberhinaus werden auch weiterführende Konzepte von Rust veranschaulicht.

3.4 Grundlegende Datenstruktur

Eine erste Idee, wie die Datenstruktur aussehen könnte, war ein einfacher binärer Baum, dessen Knoten die Operanden und die Blätter Einzelne Werte sind. Dies entspricht dem abstrakten Syntaxbaum nach dem Parsen. Dies könnte beispielsweise für den Term $x + 1 + 2$ folgendermaßen aussehen.



In Rust sähe die Implementierung folgendermaßen aus. Die Ausprägung Add benötigt für die Werte immer den Wrapper `Box` der die Daten auf den Heap speichert, da sich zur Compilezeit nicht die Größe auf dem Stack bestimmen lässt. Konkret kann der Compiler nicht bestimmen wie viele Ebenen Add beinhaltet.

Codebeispiel 20: BTree Ast

```

enum BTree {
    Number(i64),
    Symbol(String),
    Add(Box<BTree>, Box<BTree>),
}

let tree = BTree::Add(
    Box::new(BTree::Add(
        Box::new(BTree::Symbol("x".to_owned())),
        Box::new(BTree::Number(1)),
    )),
    Box::new(BTree::Number(2)),
);

```

Bei der direkten Änderung bzw. Zusammenfassung des Baums ergibt sich nun durch das Ownership ein Problem. Man bräuchte auf jeden Knoten eine veränderbare Referenz und müsste dann von unten nach oben immer prüfen, welche Blätter sich zusammenfügen lassen. Diese müssten dann entfernt werden und vorherige Elternknoten müssten zu Blättern umgewandelt werden.

Würde versucht werden die Referenzen in einem Vektor zu speichern ist das zumindest bereits für Add nicht möglich. Da `Box` das Ownership des Wertes nimmt kann dazu parallel keine veränderbare Referenz existieren. Dasselbe Problem tritt auf, wenn die Blätter bereits

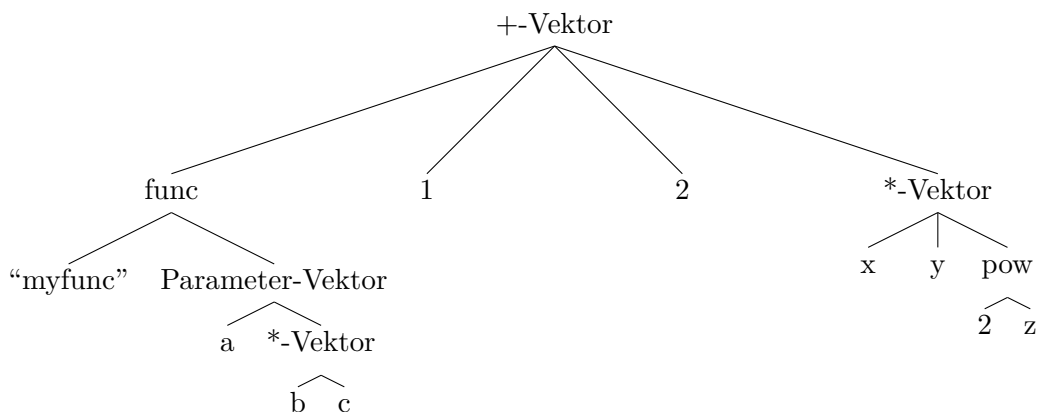
der entsprechende `Box`-Wert sind, dann würde das Ownership immer noch an `parent_1` übergeben werden. Hier könnte nun versucht das Problem mit Referenzzählern (`Rc` und `RefCell`) zu lösen, die speziell dafür gedacht sind. Auch wäre wahrscheinlich die Nutzung von Rohzeigern in Verbindung mit `unsafe` möglich, was Code in Rust kennzeichnet, der evtl. auf undefinierten Speicherbereich zeigen kann.

Codebeispiel 21: get refs

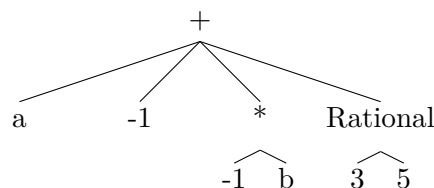
```
let mut leave_1 = BTree::Number(1);
let mut leave_2 = BTree::Number(2);
let mut parent_1 = BTree::Add(Box::new(leave_1), Box::new(leave_2));

let mut refs = vec! [&mut parent_1, &mut leave_1, &mut leave_2];
println!("{:?}", refs);
```

Eine Alternative wäre nur lesende Referenzen zu erzeugen, über diese zu iterieren und dann einen neuen Baum zu erstellen. Wenn aber sowieso alle Werte benötigt werden um diese zusammenfassen zu können, ist es eine bessere Idee, Werte, die Kommutativ sind, in einem Vektor in einer Ausprägung des Enums um auf diese Zugreifen zu können. Für Addition und Multiplikation würde dann der Baum für den Term $myfunc(a, b * c) + 1 + 2 + x * y * 2^z$ beispielsweise so aussehen:



Werden die Bedingungen und Vereinfachungen noch angewendet, so ergibt sich für den Term $a - (1 + b) + 3/5$ der Baum:



Diese Struktur findet sich ebenfalls in Quelle Rotes Buch. SymPy verwendet eine ähnliche Struktur ([link zu Quelle](#)). Da sich diese Struktur anscheinend für ein CAS eignet, soll diese auch in diesem System verwendet werden. Hierrauf bauen alle weiteren Operationen auf, wie z.B. das Zusammenfassen des Terms oder Substitution.

3.5 Grundtypen des CAS

In diesem Kapitel soll definiert werden, welche `structs`, `traits` oder `enums` benötigt werden, um das CAS zu erstellen. Darüberhinaus soll grob umrissen werden, welche Implementierungen im weiteren Verlauf sinnvoll sein können und welchen Zweck diese erfüllen. Es ist noch zu erwähnen, dass hier nicht definiert werden sollen, mit welchen `Derive macros` diese Typen erweitert werden sollen. In Rust ist es üblich `structs` und `enums` mit diesen Makros zu

erweitern, um z.B. anzugeben, dass der Typ kopiert oder sortiert werden kann. Die Implementierung dieser Funktionalität wird dann teilweise zur Compilezeit vom Makro übernommen.

enum PrimNum und enum PrecisionNum Diese beiden `enums` sollen als Implementierung für den Generic-Parameter im CAS dienen. Ebenso bieten diese den Vorteil, dass später ein allgemeiner Nummern-Typ existiert mit dem gerechnet werden kann, und nicht ständig manuell Type Castings durchgeführt werden müssen. So ist in Rust nicht möglich eine Addition oder Multiplikation von `i32` und `f64` durchzuführen. Hierzu muss einer der beiden Werte erst in den anderen manuell überführt werden.

Ein weiterer Vorteil ist, dass direkt eine rationale Ausprägung des `enum` implementieren werden kann. Auch hier kann der Typ direkt die Rechnung übernehmen. Hierzu werden die Operatoren `std::ops::Add` und `std::ops::Mul` für den Enum selbst überladen. Im Hinblick auf die Bedingung, dass eine Subtraktion eine Addition ist, wird ebenso `std::ops::Mul` für einen Ganzzahlwert mit den Enums überladen. Ebenso kann es von Interesse sein, ob es sich um das neutrale Element der Addition oder Multiplikation handelt um einen Term zu vereinfachen. Für diesen Fall wird der Trait `PartialEq` für Ganzzahlen überladen, um hier nach Bedarf auch andere Werte zu überprüfen. Eine Alternative wären Methoden, die einen `bool`-Wert zurückgeben, ob es sich um einen solchen Wert handelt.

In unserer Implementierung wird `PrimNum` solange wie möglich mit Ganzzahlen und Rationalen Werten arbeiten. Sobald ein Fließkommawert addiert wird, werden die beiden anderen Werte zu einer Fließkommazahl umgewandelt. Dies geht zulasten der Genauigkeit und könnte bei Bedarf so angepasst werden, dass ein Rationaler Typ erzeugt wird. Es wäre auch denkbar, Fließkommazahlen und andere numerische Typen nicht standardmäßig zusammenzufassen.

trait NumberType Dieser Trait soll hauptsächlich als Typparameter oder als Einschränkung für Generics zum Einsatz kommen. Wie bereits erwähnt soll ein Teil des CAS mit verschiedenen Typen durchführbar sein und alle Typen, die diesen Trait implementieren können dann für das CAS genutzt werden. Im konkreten Fall werden dies die beiden `enums` sein, die gerade erwähnt worden sind. Zudem werden Funktionen als Bedingung definiert um die neutralen Elemente der Addition und der Multiplikation zurückzugeben als auch eine Möglichkeit einen Bruch zu kürzen. Diese müssen dann von den `enums` implementiert werden.

Außerdem kann dieser Trait bereits weitere Bedingungen aufnehmen wie z.B. dass der implementierende Enum gewisse Operatorenüberladungen haben muss oder auch, dass die Werte vergleichbar und sortierbar sind.

struct Parser Auch wenn angedacht ist, eine bestehende Bibliothek zur lexikalischen Analyse von Strings einzusetzen, benötigen wir einen Parser, der die generierten Tokens oder die entstehende Datenstruktur in die Struktur des Systems überführt.

3.6 Datenstruktur in Rust

Mit dem Wissen über Generics und den theoretischen Überlegungen zur Datenstruktur, kann diese in Rust folgendermaßen umgesetzt werden:

Codebeispiel 22: Datenstruktur in Rust

```
pub enum Ast<N> {
    Add(Vec<Ast<N>>),
    Mul(Vec<Ast<N>>),
    Pow(Box<Ast<N>>, Box<Ast<N>>),
    Symbol(String),
```

```

    Const(String),
    Func(String, Vec<Ast<N>>),
    Num(N),
}

```

Add und Mul Diese beiden Ausprägungen besitzen als Wert einen Vektor, der weitere Elemente enthält. So sind diese einfach abrufbar und der Vektor kann gezielt geändert werden, z.B. zum Sortieren der Werte oder das Entfernen von unnötigen Elementen (neutrale Elemente der Addition oder Multiplikation).

Pow Diese Ausprägung besitzt zwei Parameter, Basis und Exponent, die jeweils wieder beliebige Ausdrücke sein können.

Symbol Ein einfacher String, der einen Variablennamen enthält. Diese sollen später einfach substituiert werden können.

Const Ein einfacher String, der einen Konstantennamen enthält, auch in UTF-8 möglich, z.B. π . Eigens hinzugefügte Funktionen können somit prüfen, um welche Konstante es sich handelt und entsprechend eine Vereinfachung implementieren. Es besteht darüberhinaus die Möglichkeit Konstanten als Wert zurückzugeben.

Func Diese Ausprägung besitzt ebenfalls zwei Parameter, den Namen der Funktion und die übergebenen Parameter als Vektor. Der Name kann somit einfach ausgewertet, die Parameter einfach übergeben werden. Beide Werte werden an die dynamisch hinzugefügten Auswertungsfunktionen übergeben.

Num Diese Ausprägung enthält als Wert eine Variante, die das `trait NumberType` implementiert.

Auswertung des Terms Eine Hauptfunktionalität, die dieses Enum erhält, ist die Möglichkeit einen Term zusammenzufassen und falls möglich soweit wie möglich in ein numerisches Ergebnis zu überführen. Dabei muss natürlich einerseits die Ausführungsreihenfolge beachtet werden, andererseits soll diese Methode leicht erweiterbar sein.

In unserem System wird die Methode rekursiv implementiert. Eine iterative Lösung wäre ebenfalls möglich, aufgrund des Ownership und Borrowing in Rust aber tatsächlich recht komplex und weniger leserlich und verständlich. (Quelle? Beispiel wie rekurs to iterativ in Rust -> sehr codelastig). Da Rust keine weiteren Besonderheiten für Rekursion bietet, wird diese Implementierung ebenfalls verbesserungswürdig sein im Hinblick auf die Geschwindigkeit.

Bei der entsprechenden Auswertung einer Ausprägung des `Ast` werden dabei erst alle "Kindelemente" ausgewertet. Für die Addition bedeutet dies konkret, dass erst alle Elemente im Vektor ausgewertet werden und danach erst versucht wird zu addieren. Für Potenzen werden entsprechend erst Basis und Exponent ausgewertet.

Eine erste Implementierung hierfür sieht folgendermaßen aus. Die beiden zusätzlichen Parameter werden noch näher in den Kapiteln zu "Erweitbarkeit in Rust" und "Ungenauigkeiten erlauben" näher erläutert.

Bei der Entscheidung, ob das Element direkt bearbeitet wird oder ein neuer `Ast` zurückgegeben wird, wurde der Definition von SymPy gefolgt, dass der originale Term nicht verändert wird und ein neuer erstellt wird. Auch ist hier ersichtlich, dass teilweise die `clone`-Methode aufgerufen werden muss, da ansonsten ja nur die Referenz `self` zurückgegeben werden würde.

Der `Copy`-Trait kann für `Ast` nicht implementiert werden, da bereits Vektoren diesen Trait nicht implementieren.

Codebeispiel 23: eval-Methode 1. Implementierung

```
pub fn eval(
    &self,
    evaler: &EvalFn<N>,
    hard_eval: &bool,
) -> Ast<N> {
    match self {
        Ast::Add(vec) => add(
            vec.iter()
                .map(|t| t.eval(evaler, hard_eval))
                .collect(),
            evaler,
            hard_eval,
        ),
        Ast::Mul(vec) => mul(
            vec.iter()
                .map(|t| t.eval(evaler, hard_eval))
                .collect(),
            evaler,
            hard_eval,
        ),
        Ast::Pow(base, exp) => pow(
            base.eval(evaler, hard_eval),
            exp.eval(evaler, hard_eval),
            evaler,
            hard_eval,
        ),
        Ast::Func(name, args) => func(
            name,
            args.iter()
                .map(|t| t.eval(evaler, hard_eval))
                .collect(),
            evaler,
            hard_eval,
        ),
        Ast::Const(name) if *hard_eval => {
            if evaler.consts.contains_key(name) {
                evaler.consts[name]()
            } else {
                self.clone()
            }
        }
        _ => self.clone(),
    }
}
```

Allgemeine Anmerkungen Der Generic-Typparameter `N` soll den gewählten Nummern-typ darstellen. Dieser wird aber noch nicht eingeschränkt, da sonst diese Einschränkungen später für alle implementierten Blöcke gelten muss. Dies kann z.B. nicht gewünscht sein, wenn eine Grundlegend andere Art implementiert werden soll um Terme zu verarbeiten, bei der der Trait `NumberType` keine Verwendung finden soll. Mit diesem Ansatz können wir auch die definierte Ausführungsreihenfolge der Ausprägungen bestimmen. Bevor also `Ast::Add` ausgewertet wird, werden zuerst alle Elemente des Vektors ausgewertet. Eventuelle vereinfachungen sind dann bereits vorgenommen.

Datenstruktur auf dem Speicher Mit dieser Implementierung wird ein Großteil der Daten auf dem Heap abgelegt. Einzig die Variante `Ast::Num<PrimNum>` kann komplett auf dem Stack abgelegt werden, da diese eine definierte Größe hat. String und Vec speichern die Daten auf dem Heap und halten nur einen Zeiger auf dem Stack falls es das oberste Element ist. Box legt die Daten wie bereits besprochen aus Gründen der ungewissen Speicheranforderung auch auf dem Heap an, lediglich der Zeiger kann sich auf dem Stack befinden.

Zwei konkrete Beispiele sollen dies veranschaulichen. Der String bei `Symbol(x)` wird intern als Vektor mit dem Typ `u8` repräsentiert. ([link quelle](#))

Codebeispiel 24: Term 1 Stack und Heap

```
let term = Ast::Num(PrimNum::Int(42));
Stack:
Ast::Num
+-----+
| PrimNum::Int |
| +-----+   |
| | 42 |     |
| +-----+   |
+-----+
```

Codebeispiel 25: Term 2 Stack und Heap

```
let term = Ast::Add(vec![
  Ast::Num(PrimNum::Int(1337)),
  Ast::Symbol("x".to_owned()),
  Ast::Mul(vec![
    Ast::Num(PrimNum::Float(1.1)),
    Ast::Pow(
      Box::new(Ast::Num(PrimNum::Int(2))),
      Box::new(Ast::Num(PrimNum::Int(2))),
    ),
  ]),
]);
Stack:
Add(Vec) (Beispielhaft, hier fehlt z.B. noch die capacity)
+-----+-----+
| ptr | len |
+---+-----+
v
Heap:
Num(Int)  Symbol(Vec<u8>)  Mul(Vec)
+-----+ +-----+ +-----+
```

1337	ptr	len	ptr	len	
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
	v		v		
	+-----+	Num(Float)	Pow		
	120	+-----+	+-----+		
	+-----+	1.1	ptr	ptr	
		+-----+	+-----+	+-----+	
			v	v	
			Num(Int)	Num(Int)	
			+-----+	+-----+	
			2	2	
			+-----+	+-----+	

4 Parser für mathematische Ausdrücke

Ziel dieses Kapitels ist die Nutzung bzw. Implementierung der lexikalischen Analyse bzw. eines Parsers um Terme in das Enum `Ast` zu überführen. Erst hier sollen Vereinfachungen und Zusammenfassungen gemacht werden.

4.1 Prüfung von bestehenden Bibliotheken zum Parsen

mexprp Rust (quellen) Diese Bibliothek wurde gewählt, da einerseits beim Parsen eine beliebige Genauigkeit möglich ist. Andererseits schien nach einem kurzen Blick in den Code es auch möglich den Term einfach in unsere gewünschte Form zu bringen, da das Enum `Term` anscheinend einen Abstract Syntax Tree implementiert wie am Anfang angedacht. Über diesen müsste nur einmal iteriert werden um zu unserer modifizierten Version zu gelangen.

Die Abhängigkeit “rug” soll die beliebige Genauigkeit ermöglichen. Dabei handelt sich um ein Interface zu den GNU-Bibliotheken GMP, MPFR, MPC. Für diese müssen aber unter GNU/Linux, macOS und Windows jeweils andere Abhängigkeiten installiert werden. Ein schnelles testen auf anderen Rechnern ist somit ausgeschlossen. Aus diesem Grund wird diese Bibliothek nicht verwendet. Ebenso wird deshalb “rug” nicht als Bibliothek für die beliebige Genauigkeit verwendet, sondern num und bigdecimal.

meval Rust Diese Bibliothek erscheint recht unkompliziert und einfach zu nutzen. Die geparsten Tokens werden als UPN (umgekehrte polnische Notation) ausgewertet. Dies könnte man prinzipiell dazu nutzen unseren modifizierten Ast zu erzeugen. Ein kleiner Nachteil ist, dass nur Fließkommazahlen ausgewertet werden. Sollte man den Lexer irgendwann erweitern wollen, sodass eine beliebige Genauigkeit erreicht werden soll, müsste man diesen Part neu schreiben. Zudem habe ich diese Bibliothek leider nicht bei meiner initialen Recherche in Betracht gezogen und nicht weiter überprüft. Erst während der Entwicklung habe ich Interessehalber weiter reingesehen und die Enums `Operation` und `Token` größtenteils übernommen. Hätte ich den Lexer nicht bereits zu einem großen Teil implementiert, hätte ich die Möglichkeit genutzt, die Tokens aus dieser Bibliothek zu erzeugen.

evalexpr Rust Diese Bibliothek erscheint sehr Umfangreich. Allerdings lässt sich eigentlich nur die `eval`-Funktion nutzen, da fast alle anderen Strukturen nicht mit dem Schlüsselwort `pub` versehen sind. Es war mir daher nicht möglich, den Tokenizer bzw. den Tree direkt zu nutzen, in dem sich ein bereits geparster String befindet. Daher schied diese Bibliothek ebenfalls aus.

tinyexpr C Da es in Rust auch möglich ist, bereits bestehenden C-Code zu nutzen, wurde ebenfalls geprüft eine einfache C-Bibliothek zu nutzen, sodass auch das die notwendige Schnittstelle des Parsers nicht allzu kompliziert wird. Hier habe ich mich für `tinyexpr` entschieden. Leider mussten Variablen bereits vor der lexikalischen Analyse zumindest bekannt sein (siehe Readme <https://github.com/codeplea/tinyexpr> te compile). Da ich mich generell gegen diesen Schritt entschieden habe, wurde auch diese Bibliothek nicht verwendet.

Fazit Eine geeignete Bibliothek zu finden, die den Anwendungszweck erfüllt und dabei nicht allzu weit weg vom eigentlich Ziel ist, ist leider in meinen Augen nicht möglich gewesen. Daher wurde sowohl die lexikalische Analyse als auch das Parsing selbst implementiert. Die wichtigsten Eigenschaften hierzu habe ich aus dem Kurs 01810 Übersetzerbau herausgearbeitet, den ich im Semester xx belegt habe.

4.2 Implementierung der lexikalischen Analyse bzw. Tokenizer

In diesem Schritt soll nur aus einer übergebenen Zeichenkette ein Vektor mit Tokens erstellt werden. Hierbei wird Zeichenweise gearbeitet und nicht über reguläre Ausdrücke. Eine Besonderheit in Rust, dass ein Char in einem String nicht über einen Index angesprochen werden kann. Da Strings als UTF-8 gespeichert werden, ist mit dem Index nicht eindeutig, ob man das entsprechende Byte haben möchte oder das entsprechende Zeichen. Daher muss der String hier erst in einen Vektor des Typ `Char` umgewandelt werden.

Die Tokens werden dabei wieder als Enum implementiert. In diesem Abschnitt verzichten wir auf Generics und implementieren den `struct` nur für den Typ `PrimNum`.

Im Enum benötigen wir folgende Werte:

Codebeispiel 26: Enum Token

```
pub enum Token {
    LParen,
    RParen,
    Comma,
    Var(String),
    Func(String),
    Op(Operator),
    Num(PrimNum),
}
```

LParen und RParen - linke und rechte Klammer Diese dienen dazu um Terme zu gruppieren und die eigentliche Operatorenreihenfolge zu verändern, wie es üblich ist in der Mathematik. Es werden die Zeichen “(” und “)” erkannt.

Comma Dient als Separator von Funktionswerten. Fließkommazahlen werden nur im englischen Format mit Punkt akzeptiert.

Var und Func Eine alphanumerische Zeichenkette, die zwingend mit einem Buchstaben beginnt. Klein- und Großschreibung sind gleichermaßen erlaubt, die Namen beachten die Groß- und Kleinschreibung. Eine Funktion wird daran erkannt, dass nach der Zeichenkette eine öffnende Klammer “(” anschließt, die zwingend wieder geschlossen werden muss.

Op Mögliche Operatoren, es wird Beschränkt auf “+, -, *, /, ^”. Der Typ Operator ist dabei wieder ein Enum, der diese Einträge enthält. Im Lexer werden noch keine Anpassungen des Terms vorgenommen.

Num Eine Zahl, eine Fließkommazahl wird daran erkannt, dass das erste Zeichen ein “.” ist oder die gesamte geparsete Zeichenkette einen “.” enthält. Es sind nur Zahlen erlaubt.

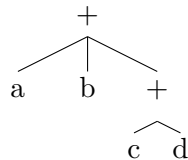
4.3 Implementierung Parser

Als Parser kommt eine einfache Implementierung der Operator-Vorranganalyse in Frage, wie Sie in Kurs 1810 ke3 3.3.2 beschrieben ist. Es handelt sich also um eine Bottom-Up-Analyse. Der zu implementierende Parser orientiert sich an der beschriebenen “Precedence climbing method” aus dem Buch “BCPL, the language and its compiler”. Dabei wird der Algorithmus so verändert, sodass der momentan gültige Operator in einer `while`-Schleife läuft, um einen Vektor befüllen zu können. Die “linke” Seite des Operators wird dabei jeweils vor der Schleife

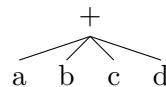
geparst, die “rechten” Seiten dann in der Schleife. Dazu werden auch gleich Operatorenüberladungen des **Ast**-Enum verwendet.

Zudem werden zusätzlich noch Funktionsaufrufe beim Parsen hinzugefügt. Die jeweiligen Parameter fangen dabei wieder mit der Funktion zum Parsen von “+” und “-” an. Auch werden bereits Variablen und Konstanten unterschieden.

Darüberhinaus werden entsprechende Prüfungen eingebaut, sodass der Term $a+b+(c+d)$ nicht wie der folgende Baum geparkt wird:



Es entsteht vielmehr gleich der Baum:



Weitere Anmerkungen zu dem Kurs 01810: Es handelt sich um einen LR(1)-Parser, da der Parser ein Zeichen vorrauschen kann um die nächste Aktion zu bestimmen. Der Stack ist im weitesten Sinne bereits ein gültiges **Ast**-Element, das befüllt wird. Die **shift**-Funktion kann als Voranschreiten im Iterator angesehen werden, was immer dann passiert, wenn ein Operator oder ein Terminal gelesen wird (vgl. Grammatik). Die **reduce**-Funktion entspricht dem Anwenden der Operatorenüberladen auf dem gültigen **Ast**-Element.

Mögliche Verbesserungen dann im Fazit (z.B. erlauben von $3x$ statt $3*x$ und Parser so umbauen, dass dynamisch Operationen hinzugefügt werden können, z.B. Fakultät anstelle der Nutzung von Funktionen, auch wenn wahrscheinlich umgeschrieben in Funktion um dann wieder evaluieren zu können.)

Die Grammatik dazu sieht folgendermaßen aus: (siehe kurs 1810 ke2 Seite 6)

Hier dann noch ordentlich die Grammatik von papier in Latex übertragen! Da probiere ich verschiedenes aus mit dem Mathemodus, da ich latex nicht oft benutze.

5 Grundfunktionlitäten

In diesem Abschnitt sollen Grundlegende Funktionalitäten des CAS implementiert werden. Dies betrifft sowohl den Aufbau des Codes in Rust, um das CAS einfach erweiterbar zu machen, als auch das grundsätzliche Rechnen des Systems. So wird es beispielsweise keine Standardfunktionalität sein, dass $\cos(x)^2 + \sin(x)^2 = 1$ erkannt wird. Dies soll über eine Erweiterung erkannt und verarbeitet werden. Das Zusammenfassen von $x + x = 2 * x$ ist hingegen aber eine Grundfunktionlität.

Ähnlich wie in SymPy soll der auszuwertende Ausdruck nicht verändert werden, sondern ein neuer zurückgegeben werden. Dies vereinfacht die Implementierung enorm, hat aber natürlicherweise den Nachteil zusätzlichen Speicherplatz zu belegen.

5.1 Änderung des Terms mithilfe Operatorenüberladung

Der Trait `NumberType` soll bereits die meisten nötigen Operatorenüberladungen von den zu implementierenden Typen verlangen. Die wichtigsten Operationen werden “Add” und “Mul” sein, da diese vom CAS benutzt werden. Operatoren die ebenfalls interessant im weiteren Verlauf sein können sind der Modulo-Operator (Auswertung der sin- oder cos- Funktion), aber auch die Möglichkeit Werte vergleichen zu können. Deshalb soll noch `PartialEq` und `PartialOrd` implementiert werden um die Funktionen “`NumberType == 1`” und “`NumberType > 1`” zu ermöglichen. Daraus ergibt sich die Definition und die Implementierungen für `PrimNum`:

Codebeispiel 27: trait `NumberType` 1. Iteration

```
pub trait NumberType:
+ PartialEq<i128>
+ PartialOrd<i128>
+ ops::Add<Self, Output = Self>
+ ops::Mul<Self, Output = Self>
+ ops::Rem<i128, Output = Self>
{}

impl PartialEq<i128> for PrimNum { ..., }
impl PartialOrd<i128> for PrimNum { ... }
impl ops::Add<PrimNum> for PrimNum { ... }
impl ops::Mul<PrimNum> for PrimNum { ... }
impl ops::Rem<i128> for PrimNum { ... }
```

Ebenso soll für das Enum `Ast` ebenfalls Operatoren überladen werden. Hier beschränken wir uns auf die Operatoren “Add” und “Mul”. Mit dem zusätzlichen Generic-Typ ergibt sich die Definition:

Codebeispiel 28: Ast Operatorenüberladung

```
impl<N> ops::Add<Ast<N>> for Ast<N>
where
    N: NumberType,
{ ... }

impl<N> ops::Mul<Ast<N>> for Ast<N>
where
    N: NumberType,
```

```
{ ... }
```

Die Überladungen für `Ast` sollen dabei gleich Zusammenfassungen übernehmen falls möglich. Das bedeutet konkret für Addition (und entsprechend für Multiplikation):

Codebeispiel 29: smarte Operatenüberladung

```
Ast::Num(1) + Ast::Num(2) = Ast::Num(3)

Ast::Add(vec![Ast::Symbol("a"), Ast::Symbol("b")]) +
Ast::Add(vec![Ast::Symbol("c"), Ast::Symbol("d")]) =
    Ast::Add(
        vec![Ast::Symbol("a"), Ast::Symbol("b"),
            Ast::Symbol("c"), Ast::Symbol("d")]
    )

Ast::Add(vec![Ast::Symbol("a"), Ast::Symbol("b")]) +
Ast::Num(1) =
    Ast::Add(
        vec![Ast::Symbol("a"), Ast::Symbol("b"),
            Ast::Num(1)]
    )

Ast::Add(vec![Ast::Symbol("a"), Ast::Num(2)]) +
Ast::Num(1) =
    Ast::Add(
        vec![Ast::Symbol("a"), Ast::Num(3)]
    )

Ast::Add(vec![Ast::Symbol("a")]) +
Ast::Anderes =
    Ast::Add(
        vec![Ast::Symbol("a"), Ast::Anderes]
    )

Ast::Anderes) +
Ast::Anderes =
    Ast::Add(
        vec![Ast::Anderes, Ast::Anderes]
    )
```

5.2 Addieren und Multiplizieren (Standardauswertung von Termen)

Die prinzipielle Vorgehensweise um Additionen und Multiplikationen auszuführen orientiert sich an dem Vorgehen aus SymPy. (entsprechender Codeabschnitt aus den Überladungen). Beide Verfahren sind im Grunde identisch, nur ändert sich der entsprechende Operator und das Ergebnis. So wird aus $x + x$ eine Multiplikation der Form $2 * x$. Aus $x * x$ wird die Potenz x^2 .

Das Verfahren wird exemplarisch für die Addition beschrieben, die Multiplikation wird aber ebenfalls im System implementiert. Kernstück ist eine HashMap der Form

`HashMap<Ast<N>, Ast<N>`. Key und Wert sind hierbei eigenständige Terme. Der Key ist hierbei der eigentliche Term, der in der Addition auftritt. Der Wert ist sozusagen der Multiplikator wie häufig der Term auftritt. Zudem gibt es einige Sonderfälle. Als Beispiel dient der Term $2*x*y+x*y+3+a-a$. Um diesen zusammenzufassen wird über alle Elemente des Vektors iteriert. Die HashMap ist zu Anfang leer, ein gesondertes “Zahlenergebnis” wird auf 0 gesetzt.

- 1. Wert: Die Multiplikation ist gleich ein Sonderfall. Hier wird versucht eine Zahl aus dem Subterm zu extrahieren, dies ist hier möglich mit der 2. Die HashMap erhält den Eintrag “[x*y] = 2”
- 2. Wert: Wie davor wird wieder versucht eine Zahl zu extrahieren, was hier nicht gelingt. Bei der Multiplikation können wir also somit das neutrale Element 1 für diesen Subterm hinzufügen. Die HashMap erhält nun den aktualisierten Wert “[x*y] = 3”
- 3. Wert: Alle Zahlen, also unser definierter Nummerntyp, und falls es sich bei dem Subterm um eine Addition handelt, können einfach zu dem Zahlenergebnis addiert werden. Hierzu wird einfach die bereits vorhandene Operatorenüberladung genutzt
- 4. Wert: Alle anderen Ausprägungen des `Ast`-Enum (in diesem Fall `Symbol`) werden ähnlich der Multiplikation behandelt, nur wird hierbei immer der Wert 1 addiert. Die HashMap erhält hier nun den Wert “[a] = 1”
- 5. Wert: Augenscheinlich wieder ein `Symbol`. Allerdings wurde bereits erwähnt, dass diese Form intern als $+(-1 * a)$ gespeichert wird. Also wird zur HashMap wieder eine Multiplikation hinzugefügt, die den Multiplikator “-1” hat. Somit wird die HashMap aktualisiert mit “[a] = 0”

Nachdem diese Vereinfachungen in der HashMap gespeichert sind, besteht die Möglichkeit auf dieser HashMap durch externe Funktionen noch weitere Vereinfachungen vorzunehmen. Dieses Vorgehen wird noch genauer im Kapitel “Erweiterbarkeit in Rust und Beispiele” näher erläutert.

Zum Schluss können die Ergebnisse wieder in die Ausprägung `Ast::Add` überführt werden. Das Zahlenergebnis kann hierbei einfach hinzugefügt werden. Bei der HashMap hängt dies vom Wert zum Key ab.

- Wert == 0: Key wird dem Vektor nicht hinzugefügt
- Wert == 1: Key wird dem Vektor hinzugefügt
- Alles andere: Es wird dem Vektor “`Ast::Mul(vec![Wert, Key])`” hinzugefügt

Um dieses Verfahren zu implementieren, muss für das `Ast`-Enum noch die Traits `Eq`, `PartialEq` und `Hash` implementiert werden. Erst dann kann der Typ `Ast` als Key in der Map verwendet werden. Im einfachsten Fall, kann dies per Makro geschehen. In unserem geht dies aber nur für `PartialEq`. Da z.B. Vektoren die Traits `Eq` und `Hash` nicht implementieren, müssen wir dies selbst erledigen. Der `Eq` ist dabei aber nur deklarativ, es muss keine Funktion ausgefüllt werden, dies übernimmt der Compiler (<https://doc.rust-lang.org/std/cmp/trait.Eq.html>). Der `Hash`-Trait kann ebenfalls recht einfach implementiert werden, hier können wir alle vorhandenen Werte als Hash zusammenfassen.

Ideen zur Verbesserung Es ist zu überlegen, den Nummerntyp nicht zusammenzufassen, sodass Rationale Zahlen und Fließkommazahlen nebeneinander existieren. In der momentanen Implementierung wird diese Addition zu einem Fließkommazahlen vereinfacht. Der Vorteil dabei ist, dass sicher nur ein Zahlenwert vorhanden. Allerdings wird der Nachteil der Ungenauigkeit in Kauf genommen.

Bei der Zusammenfassung in der HashMap kann es natürlich zu Konflikten kommen. Eine andere Idee wäre hier als Key den Term als String zu verwenden. Hierzu müsste aber der Term auf jeden Fall vorher sortiert sein. Permutationen eines Terms ($x * y$ und $y * x$) können sonst nicht abgefangen werden. Hier hat das Hashing den Vorteil, das die Reihenfolge egal ist, da beide Symbole zusammen denselben Hashwert errechnen.

5.3 Umgang mit Brüchen, Potenzen und Wurzeln

Um auch genauere Werte zu erlauben, sollen rationale Zahlen in Form von Brüchen implementiert werden. So können mithilfe der Operatorenüberladungen diese dann zusammengefasst und gekürzt werden. Für rationale Zahlen sind im Enum `PrimNum` für Zähler und Nenner nur ganze Zahlen erlaubt, sowohl jeweils positiv und negativ. Der Term $1/3 + 1/3$ wird also zu `Num(Rational(2, 3))` ausgewertet. Der Term $1/3 + 1/3 + 1/3$ soll zu `Num(Int(1))` ausgewertet werden können.

Terme wie $1/x$ werden durch das CAS in die Form `Pow(Symbol("x"), Num(Int(-1)))` umgeformt. Dabei ist "x" entsprechend die Basis und "-1" der Exponent. In der Standardimplementierung soll es außerdem bereits möglich sein die Potenzregel $a^{b^c} = a^{(b*c)}$ anzuwenden. Durch die bisherige Implementierung der Multiplikation wird außerdem bereits die Potenzregel $x^a * y^a = x^{(a+b)}$ angewendet. Die letzte Regel $x^a * y^a = (x * y)^a$ soll wiederum als erweiterte Funktionalität implementiert werden, da hier eine andere Vorgehensweise bei dem Zusammenfassen des Terms notwendig ist, als die bisher implementierte und in Kapitel x vorgestellte.

Glücklicherweise können Wurzeln ebenfalls als Potenzen dargestellt werden, so ist z.B. $\sqrt{2} = 2^{1/2}$ oder $\sqrt[3]{8} = 8^{1/3}$. So können alle bisherigen Regeln hierrauf einfach angewendet werden. Zudem sollen die Funktionen `sqrt` und `nthroot` im Term verwendet werden können, die dann entsprechende `Ast`-Elemente erzeugen.

Darüberhinaus wird derselbe Algorithmus (Newton's Method) wie in SymPy implementiert um zu prüfen, ob eine Wurzel eine perfektes Ergebnis liefert. In diesem Fall kann das entsprechende Element vereinfacht werden.

5.4 Erweiterbarkeit in Rust und Beispiele

Um die Erweiterbarkeit des Systems zu gewährleisten wird das Struct `EvalFn` eingeführt. Ein Objekt des struct soll zur Evaluierung eines Terms mit angegeben werden. Die hinzugefügten Funktionen des entsprechenden Objekts können dann nach allen Standardoperationen den Term weiter vereinfachen, wie z.B. den angesprochenen Sonderfall $\cos(x)^2 + \sin(x)^2 = 1$, und Konstanten falls gewünscht auswerten. Außerdem wird ein Standard-Objekt vom System bereitgestellt, dass bereits die gängigsten Vereinfachungen vornehmen kann.

Codebeispiel 30: Definition EvalFn

```
pub struct EvalFn<N> {
    pub adders: Vec<fn(&mut HashMap<Ast<N>, Ast<N>), &bool)>,
    pub muls: Vec<fn(&mut HashMap<Ast<N>, Ast<N>), &bool)>,
    pub pows: Vec<fn(&Ast<N>, &Ast<N>, &bool) -> Option<Ast<N>>>,
    pub funcs:
        HashMap<String, fn(&Vec<Ast<N>), &bool) -> Option<Ast<N>>>,
}
```

```
pub consts: HashMap<String, fn() -> Ast<N>>,
}
```

Allgemeines Da das Enum `Ast` bereits einen Generic-Parameter enthält, muss dieser ebenfalls im struct angegeben werden. Die Felder sind teils Vektoren, teils HashMaps, die entsprechende Funktionen zur Vereinfachung vorhalten. Der Parameter mit dem Typ `&mut HashMap<Ast<N>` wurde bereits genauer vorgestellt. Durch die Übergabe der Map anstelle des ausgewerteten `Ast`-Typs erhoffe ich mir eine einfachere Berechnung. So kann direkt über die Keys geprüft werden, ob ein Term enthalten ist, der ansonsten eventuell verschachtelt in einer Multiplikation liegt. (Beispiel!!)

adders Funktionen, die den Typ `Ast::Add` weiter vereinfachen können. So können entsprechend die Einträge der HashMap geprüft und verändert werden. Der zweite Parameter vom Typ `bool` gibt an, ob es sich um eine explizite Berechnung handeln soll. Dies soll noch Relevant werden, wenn auch ungenaue Werte berechnet werden sollen. Ein Beispiel ist die bereits mehrfach angesprochene Vereinfachung $\cos(x)^2 + \sin(x)^2 = 1$.

muls Funktionen, die den Typ `Ast::Mul` weiter vereinfachen können. Die Parameter sind identisch wie bereits bei `adders`. Dieses Feld ist der Vollständigkeit halber implementiert, da ich kein Beispielterm finden konnte, bei dem dies relevant ist.

pows Funktionen, die die Basis und den Exponenten erhalten und Auswerten können. So kann z.B. der Term $(x * 4)^{(1/2)}$ zu $x^{(1/2)} * 2$ vereinfacht werden. Auch hier gibt es wieder die Möglichkeit eine Explizite Berechnung zu erzwingen. Diese Funktion besitzt darüberhinaus einen Rückgabewert. Da der entsprechende `Ast` nicht direkt bearbeitet wird, da sich dieser auch in der Ausprägung verändern kann. In diesem Beispiel wird aus einem `Ast::Pow` ein `Ast::Mul`.

funcs Funktionen, die angegebene Funktionen im Term umschreiben bzw. auswerten sollen. So kann z.B. der Term $\sin(0)+2$ direkt vereinfacht werden in $0+2 = 2$. Der Term $\text{sqrt}(4)$ kann zu 2 vereinfacht werden. Dabei bleiben Funktionen, die `EvalFn` nicht bekannt sind natürlich im Term unverändert erhalten. Falls eine Auswertung nicht möglich ist, ist dies ebenfalls der Fall. Ansonsten steht der Funktion natürlich immer offen den Term entsprechend in ein anderes `Ast`-Objekt umzuwandeln. Dieses Feld ist als HashMap implementiert, da eine Funktion nur eine Auswertung besitzt. Das Überschreiben einer bereits bestehenden Funktion ist somit simpel.

consts Funktionen die nur ein `Ast` zurückgeben. Dies wird in den meisten Fällen ein numerischer Wert sein. Dieses Feld ist ebenso als HashMap umgesetzt.

5.5 Implementierung von Substitution

Um nun endlich einen Term mit Variablen auszuwerten, müssen wir diese im `Ast`-Enum substituieren können. Diese Funktionalität kann recht einfach implementiert werden. Hierzu wird die bestehende `eval`-Methode erweitert, sodass nun auch Variablen überprüft werden. Als zusätzliche Parameter erhält die Methode einen Variablennamen und den zu substituierenden Term. Sind beide gesetzt und es handelt sich bei der Auswertung um die Variable wird der Term anstelle der Variable zurückgegeben. Da Rust keine Methodenüberladung unterstützt,

werden ebenfalls Methoden implementiert, die den Funktionsaufruf auch mit weniger Argumenten unterstützen und entsprechend Standardwerte übergeben, die keinen Einfluss auf die Berechnung haben.

Die Erweiterung der Methode besitzt zwei weitere Parameter, den optionalen Variablennamen, der ersetzt werden soll, und das entsprechend [Ast](#), mit dem substituiert werden soll.

Codebeispiel 31: Erweiterung pub fn eval

```
pub fn eval_sub(
    &self,
    evaler: &EvalFn<N>,
    hard_eval: &bool,
    sub: &Option<&str>,
    with: &Option<&Ast<N>>,
) -> Ast<N> {
    match self {
        Ast::Add(vec) => add(
            vec.iter()
                .map(|t| t.eval_sub(evaler, hard_eval, sub, with))
                .collect(),
            evaler,
            hard_eval,
        ),
        Ast::Mul(vec) => mul(
            vec.iter()
                .map(|t| t.eval_sub(evaler, hard_eval, sub, with))
                .collect(),
            evaler,
            hard_eval,
        ),
        Ast::Pow(base, exp) => pow(
            base.eval_sub(evaler, hard_eval, sub, with),
            exp.eval_sub(evaler, hard_eval, sub, with),
            evaler,
            hard_eval,
        ),
        Ast::Func(name, args) => func(
            name,
            args.iter()
                .map(|t| t.eval_sub(evaler, hard_eval, sub, with))
                .collect(),
            evaler,
            hard_eval,
        ),
        Ast::Const(name) if *hard_eval => {
            let mut ret = None;
            for const_struct in evaler.consts.iter() {
                if let Some(v) = const_struct.eval(&name) {
                    ret = Some(v);
                    break;
                }
            }
        }
    }
}
```



```

        ret.unwrap_or(self.clone())
    }
    Ast::Symbol(name) if sub.is_some() && name == sub.unwrap() =>
        with.unwrap().clone(),
    _ => self.clone(),
}
}

```

Der bisherige Aufbau der `eval`-Methode stellt sicher, dass die bereits vorgestellte Ausführungsfolge von Kapitel x beibehalten wird. Zu überlegen ist noch, ob der übergebene Term vor der Substituierung ebenfalls evaluiert wird, oder ob darauf vertraut wird, dass immer ein bereits evaluierter Term übergeben wird.

Verbesserungs- und Erweiterungsideen Diese Funktionalität könnte nun beliebig erweitert werden. Einige Beispiele sind:

- Übergabe eines Vektors von Werten, Rückgabewert ist ebenfalls Vektor mit den entsprechenden Substituierungen. Diese könnten sogar Nebenläufig umgesetzt werden.
- Mehrere Variablen gleichzeitig ersetzen.
- Substitution von ganzen Termen erlauben, nicht nur einzelnen Variablen.

5.6 Implementierung von `expand`

Eventuell ist es interessant einen Term vereinfacht vorliegen zu haben. Auch wenn der Name “expand” nicht danach klingt, einen Term zu vereinfachen, so kann dies durchaus der Fall sein, wie der folgende Term zeigt $(x + 1) * (x - 2) - (x - 1) * x$ (Quelle sympy expand). Wird der Term ausmultipliziert kann dieser anschließend auf -2 vereinfacht werden. Im folgenden sollen die folgende Terme auf dieser Art vereinfacht werden können.

Multiplikation über Addition bzw. einzelne Elemente Hierbei soll der Term $x*(a+b)$ zu $x*a + x*b$ umgeformt werden, der Term $(a+b)*(c+d)*e$ soll entsprechend zu $a*c*e + a*d*e + b*c*e + b*d*e$ umgeformt werden.

Die Implementierung hiervon ist recht einfach. Bei der Ausprägung `Ast::Mul` wird ein leerer Ergebnisvektor erstellt. Danach muss über jedes Element im Vektor iteriert werden. Handelt es sich nicht um die Ausprägung `Ast::Add` wird das Element dem Ergebnis hinzugefügt falls dieser noch leer ist, ansonsten wird jedes Element im Ergebnisvektor mit dem derzeitigen Element Multipliziert, dabei können die Elemente direkt im Vektor verändert werden.

Handelt es sich jedoch um die Ausprägung `Ast::Add` wird prinzipiell mit demselben Verfahren gearbeitet. Diesmal muss nur zusätzlich über alle Elemente im Vektor von `Ast::Add` iteriert werden um das derzeitige Element zu bestimmen. Diesmal kann die Veränderung nicht direkt im Vektor erfolgen, da ansonsten bereits der Ergebnisvektor verändert wird, bevor alle Elemente mit diesem Multipliziert werden konnten.

Potenzen mit ganzzahligen positiven Exponenten Bei Potenzen der Form $(x + y + z)^n, n \in \mathbb{N}$ kann die entsprechende Basis n -mal in den Vektor für `Ast::Mul` eingefügt werden. Hiernach kann die `expand`-Funktion für `Ast::Mul` aufgerufen werden um den Term auszumultiplizieren.

Addition im Exponent Potenzen der Form n^{x+y} sollen in die Form $n^x * n^y$ überführt werden. Die Prüfung kann dabei durch eine einfache `match`-Anweisung erfolgen. Eine weitere Vereinfachung des Terms ist nach Fertigstellung nicht notwendig.

Potenzen mit Multiplikation in der Basis Potenzen der Form $(x * y)^a$ sollen in die Form $x^a * y^a$ überführt werden. Die Prüfung kann dabei durch eine einfache `match`-Anweisung erfolgen.

Beispiel der Erweiterung an der log Funktion Darüberhinaus soll es wieder einfach möglich sein, für mathematische Funktionen die `expand`-Methode aufrufen zu können. Hierzu wird das struct `EvalFn` erweitert.

Codebeispiel 32: EvalFn nach expand

```
pub struct EvalFn<N> {
    pub adders: Vec<fn(&mut HashMap<Ast<N>, Ast<N>>, &bool)>,
    pub mults: Vec<fn(&mut HashMap<Ast<N>, Ast<N>>, &bool)>,
    pub pows: Vec<fn(&Ast<N>, &Ast<N>, &bool) -> Option<Ast<N>>>,
    pub funcs:
        HashMap<String, fn(&Vec<Ast<N>>, &bool) -> Option<Ast<N>>>,
    pub consts: HashMap<String, fn() -> Ast<N>>,
    pub expand_funcs:
        HashMap<String, fn(&Vec<Ast<N>>) -> Option<Ast<N>>>,
}
```

Das neue Feld “expand_funcs” enthält Funktionen um mathematische Funktionen zu expandieren. So soll z.B. aus $\log(x^z * y)$ zu $z * \log(x) + \log(y)$ werden. Zuordnungen zu Funktionen finden wieder über eine HashMap statt.

Besonderheiten mit Rust Bei der Implementierung dieser Methode sind deutlich das strenge Typensystem von Rust als auch das Ownership und Borrowing in Erscheinung getreten. Explementarisch soll dies an zwei Beispielen erläutert werden, da sich dieses Verhalten ähnlich in anderen Fällen wiederholt.

Für das Enum `Ast` wurden die Methoden `shorten` und `sort` implementiert um generell die Handhabung des Typen zu verbessern. Beide Methoden erhalten eine mutable Referenz auf das eigene Objekt und geben auch solch eine wieder zurück. Die Implementierung von `expand` gibt allerdings eine eigenständige Instanz von `Ast` zurück, und gibt somit das Ownership in diesem Fall ab. Da `Ast` nicht den Copy-Trait implementieren kann, kann der entsprechende Rückgabewert von `shorten` und `sort` nicht einfach dereferenziert werden.

Um das Problem zu umgehen bieten sich zwei Lösungen an. Einerseits kann das zurückgegebene Objekt mit `.clone()` geklont und als eigenständiges Objekt zurückgegeben werden, oder es werden die Funktionen auf dem Objekt aufgerufen um danach das Objekt als solches zurückgegeben.

Codebeispiel 33: Mögliche Lösungen für Rückgabewerte

```
// 1. Möglichkeit
Ast::Add(result).shorten().sort().clone()

// 2. Möglichkeit
let mut result = Ast::Add(result);
result.shorten().sort();
```

```
result
```

Das zweite Beispiel bezieht sich auf explizite klonen des Exponenten bei Potenzen, wenn die Basis eine Multiplikation ist. In der entsprechenden for-Schleife wird für jedes Element des Vektors der Basis dem Ergebnis ein neues hinzugefügt nach den oben genannten Regeln. Der Exponent bleibt aber in jedem Durchlauf derselbe. Da `Ast` einerseits immer noch nicht den Copy-Trait implementiert und andererseits die Initialisierung von Box das Ownership des Wertes bekommt muss hier `.clone()` ausgeführt werden. Alle anderen `.clone()`-Aufrufe dieser Methode beheben prinzipiell dasselbe Problem.

5.7 Implementierung von simplify

6 Erweiterte Funktionalitäten (Beispiele)

Die bisherige Implementierung des CAS ist noch rudimentär. Es wurden lediglich die wichtigsten Eigenschaften implementiert um Terme möglichst weit und genau zusammenzufassen. Die weiteren Implementierungen sollen nun einen Ausblick darauf geben, wie erweiterte Funktionalitäten implementiert werden können aufgrund der geschaffenen Basis.

6.1 Terme mit Fließkommazahlen berechnen oder auch Ungenauigkeit in Kauf nehmen

Um ein einzelnes numerisches Ergebnis zu erhalten, sofern alle Variablen im Term substituiert sind, erhält das `Ast`-Enum zusätzlich die Methode `hard_eval`. Diese ruft die bereits vorgestellte Methode `eval_sub` auf und übergibt als Parameter, dass eine explizite Berechnung stattfinden soll. Dies kann mit einer gewissen Ungenauigkeit verbunden sein.

Den Parameter `hard_eval` sollen dabei alle berechnenden Funktionen - insbesondere auch später hinzugefügte Funktionen - weitergegeben werden. Dies kann durch die Typdefinition von `EvalFn` sichergestellt werden.

Als konkretes Beispiel wird die bestehende Funktion `perfect_nth_root` erweitert. Als Rückgabewert ist das Enum `Option` angegeben worden. Bisher gibt diese Funktion `None` zurück, wenn es sich um kein exaktes Ergebnis handelt. Ansonsten wird ein entsprechendes `Ast`-Element zurückgegeben.

Wenn nun `hard_eval` auf "true" gesetzt ist soll die Potenz ohne weitere Prüfung durchgeführt werden, falls es sich um Zahlen handelt. Dazu muss das trait `NumberType` noch die Methodendefinition `pow` forden, die in den eigenen Nummerntypen implementiert werden muss. In unserem Fall für `PrimNum` werden danach die entsprechenden Rust-Funktionen der primitiven Typen aufgerufen um eine Potenz zu berechnen.

Als erstes Beispiel dient der Term `sqrt(2)`. Der Parser erkennt hierbei nur die Funktion. Bei einem Aufruf von `simple_eval` wird diese Funktion dann durch die bestehende Implementierung nach `Pow(Num(Int(2)), Num(Rational(1, 2)))` umgewandelt (vgl. Kapitel Wurzel und co.). Der Aufruf von `hard_eval` dagegen gibt den Fließkommawert `1.4142135623730951` zurück, wodurch bereits eine Ungenauigkeit entstanden ist.

Als zweites Beispiel dient der Term `sqrt(2) * sqrt(2)`. Durch die gewährleistete Ausführungsreihenfolge der Terme ist das Ergebnis mit `simple_eval` 2. Dies kommt durch die Termzusammenfassung zustande, wie sie für die Addition in Kapitel x erläutert ist. Bei der Ausführung mit `hard_eval` wird dagegen entsprechend das Ergebnis `2.0000000000000004` zurückgegeben. Dies hat den Grund, da ja erst die Potenzen ausgeführt werden, und danach erst die Multiplikation durchgeführt wird. Andere Nummerntypen, die hier eine höhere Genauigkeit erlauben bzw. solche Sonderfälle abfangen, können hierbei natürlich auch immer noch ein genaueres Ergebnis liefern.

6.2 Implementierung der limit-Funktion

6.3 Implementierung einer to string-Methode

7 Vergleich zu SymPy

7.1 Korrektheit und Unterschiede gegenüber SymPy

7.2 Performancevergleich

8 Fazit

8.1 Erweiterungs- und Verbesserungsideen des Systems

8.2 Vor- und Nachteile Rust

Literatur

- [1] Jim Blandy und Jason Orendorff: *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, Inc, 1. Auflage, 2018, ISBN 978-1-491-92728-1.
- [2] Rust Book: *Performance of Code Using Generics*. <https://doc.rust-lang.org/book/ch10-01-syntax.html#performance-of-code-using-generics>. [abgerufen am 13.06.2021].
- [3] Rust Book: *The Borrow Checker*. <https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#the-borrow-checker>. [abgerufen am 13.06.2021].
- [4] Rust Book: *Using Trait Objects That Allow for Values of Different Types*. <https://doc.rust-lang.org/book/ch17-02-trait-objects.html>. [abgerufen am 13.06.2021].
- [5] Rust Book: *Validating References with Lifetimes*. <https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html>, 2021. [abgerufen am 24.07.2021].
- [6] Python docs: *Numeric Types - int, float, complex*. <https://docs.python.org/3.8/library/stdtypes.html#numeric-types-int-float-complex>. [abgerufen am 09.04.2022].
- [7] Rahul Sharma und Vesa Kaihlavirta: *Mastering RUST*. Packt Publishing Ltd., second edition Auflage, 2019, ISBN 978-1-78934-657-2.