FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Bachelor of Science Informatik

Bachelorarbeit

Ein Computeralgebrasystem in Rust

Verfasser	Bernd Haßfurther <nerglom@posteo.de>
Matrikel-Nr.	4372280
Betreuerin	Prof. Dr. Lena Oden
Datum	31. Juli 2022

Inhaltsverzeichnis

1 Einleitung **6**

2 Grundlagen CAS **7**
2.1 Grundregeln und Annahmen . . . 7
2.2 Funktionsumfang . . . 7
2.3 Überblick SymPy . . . 8

3 Vorstellung der Programmiersprache Rust **9**
3.1 Einführendes Beispiel: Ownership und Borrowing . . . 9
3.2 Was ist Rust und warum sollte es verwendet werden? . . . 11
3.3 Grundlegendes . . . 11
3.4 Enums . . . 13
3.5 Generics . . . 13
3.6 Operatorenüberladung . . . 15
3.7 Ownership, Borrowing und Lifetimes . . . 16
3.8 Stack und Heap . . . 19
3.9 Copy und Clone . . . 19

4 Implementierung des CAS **20**
4.1 Überlauf und Ungenauigkeit . . . 20
4.2 Grundlegende Datenstruktur . . . 22
4.3 Grundtypen des CAS . . . 24
4.4 Datenstruktur in Rust . . . 25

5 Parser für mathematische Ausdrücke **29**
5.1 Prüfung von bestehenden Bibliotheken zum Parsen . . . 29
5.2 Implementierung der lexikalischen Analyse bzw. Tokenizer . . . 30
5.3 Implementierung Parser . . . 30

6 Grundfunktionlitäten **32**
6.1 Änderung des Terms mithilfe von Operatorenüberladung . . . 32
6.2 Addieren und Multiplizieren (Standardauswertung von Termen) . . . 33
6.3 Umgang mit Brüchen, Potenzen und Wurzeln . . . 34
6.4 Erweiterbarkeit in Rust und Beispiele . . . 35
6.5 Implementierung von Substitution . . . 36
6.6 Implementierung von expand . . . 37
6.7 expand Implementierungshinweise . . . 38
6.8 Implementierung von simplify . . . 39
6.9 simplify Implementierungshinweise . . . 40

7 Erweiterte Funktionalitäten (Beispiele) **41**
7.1 Terme mit Fließkommazahlen berechnen und Ungenauigkeit in Kauf nehmen 41
7.2 Implementierung der limit-Funktion . . . 41
7.3 Terme leserlich darstellen . . . 42

8 Vergleich zu SymPy **43**
8.1 Korrektheit und Unterschiede gegenüber SymPy . . . 43
8.2 Performancevergleich . . . 45

9 Fazit **49**
9.1 Erweiterungs- und Verbesserungsideen des Systems 49
9.2 Ausblick 49

Verzeichnis der Codebeispiele

1 Einführendes Beispiel: struct . . . 9
2 Einführendes Beispiel: Ownership abgeben . . . 9
3 Einführendes Beispiel: Ownership zurückgeben . . . 10
4 Einführendes Beispiel: Unveränderliche Referenz . . . 10
5 Einführendes Beispiel: Veränderliche Referenz . . . 10
6 Expression Einführendes Beispiel: . . . 11
7 struct . . . 12
8 trait . . . 12
9 match . . . 13
10 enum . . . 13
11 Veranschaulichung der Monomorphisierung [39] . . . 14
12 Trait Boundaries [46] . . . 14
13 Operatorenüberladung . . . 15
14 ownership . . . 16
15 borrowing . . . 17
16 Lifetime Veranschaulichung [35] . . . 18
17 Lifetime Beispiele [47] . . . 18
18 Überlauf und Ungenauigkeit . . . 20
19 SymPy Präzision . . . 20
20 Benchmark Primitive Typen Stack Ganzzahl . . . 21
21 BTree Ast . . . 22
22 BTree Referenzen . . . 23
23 Datenstruktur in Rust . . . 25
24 eval-Methode 1. Implementierung . . . 26
25 Term 1 Stack und Heap . . . 27
26 Term 2 Stack und Heap . . . 28
27 Enum Token . . . 30
28 trait NumberType 1. Iteration . . . 32
29 Ast Operatorenüberladung . . . 32
30 Operatenüberladung für Addition . . . 33
31 Defintion EvalFn . . . 35
32 Erweiterung pub fn eval . . . 36
33 EvalFn nach expand . . . 38
34 Mögliche Lösungen für Rückgabewerte . . . 39
35 PrimNum vs. PrecisionNum . . . 43

Tabellenverzeichnis

1 Ergebnisse Benchmark Rust-Typen 21
2 Numerische Berechnungen 43
3 Terme mit Variablen zusammenfassen 44
4 Terme mit Variablen substituieren 44
5 Nutzung der Methode “expand” 45
6 Nutzung der Methode “simplify” 45
7 Performance Tests 46
8 Performance Addition 46
9 Performance Multiplikation 47
10 Speicherverbrauch 47

1 Einleitung

Computeralgebrasysteme - im Folgenden abgekürzt durch CAS - nehmen in verschiedenen Bereichen der Wissenschaft eine wichtige Rolle ein. Sie ermöglichen es mathematische Ausdrücke mit Variablen darzustellen, diese Terme zu verändern und mit diesen zu arbeiten. Andererseits lassen sich wesentlich genauere Berechnungen durchführen, indem z.B. nicht alle Ausdrücke in einem Term ausgewertet werden, wenn es hierdurch zu Ungenauigkeiten kommen kann. Die "Fachgruppe Computeralgebra" beschreibt ein solches System wie folgt:

> "Die Computeralgebra ist ein Wissenschaftsgebiet, das sich mit Methoden zum Lösen mathematisch formulierter Probleme durch symbolische Algorithmen und deren Umsetzung in Soft- und Hardware beschäftigt. Sie beruht auf der exakten endlichen Darstellung endlicher oder unendlicher mathematischer Objekte und Strukturen und ermöglicht deren symbolische und formelmäßige Behandlung durch eine Maschine. Strukturelles mathematisches Wissen wird dabei sowohl beim Entwurf als auch bei der Verifikation und Aufwandsanalyse der betreffenden Algorithmen verwendet. Die Computeralgebra kann damit wirkungsvoll eingesetzt werden bei der Lösung von mathematisch modellierten Fragestellungen in zum Teil sehr verschiedenen Gebieten der Informatik und Mathematik sowie in den Natur- und Ingenieurwissenschaften." [11]

Anwendungsbeispiele für ein CAS finden sich in der Physik, Chemie, Sicherheitstechnik, Robotik und vielen weiteren Bereichen [10]. In "Zum Einfluss von Computeralgebrasystemen auf mathematische Grundfertigkeiten" werden sowohl Vor- als auch Nachteile bei der Verwendung während der Schulzeit (z.B. wissenschaftliche Taschenrechner) diskutiert [54, S. 17 ff.]. Auch wenn somit augenscheinlich die Verwendung eines CAS viele Bereiche entlasten kann, so sollte immer noch das Hintergrundwissen vorhanden sein, welche Erleichterungen ein CAS übernimmt.

In der vorliegenden Arbeit soll ein CAS in Ansätzen mit der Programmiersprache Rust entwickelt werden. Es soll untersucht werden, inwieweit das Ownership- und Borrowing-System von Rust während der Entwicklung des CAS unterstützt und wie die Performance zur Laufzeit ausfällt. Das System soll dabei einfach erweiterbar sein. Nicht implementierte mathematische Funktionen sollen einfach integriert und Terme weiter zusammengefasst werden können als es die Standardimplentierung erlaubt.

Einen besonderen Stellenwert im Bereich der Computeralgebrasysteme nimmt SymPy ein. Dieses System ist als Open-Source-Projekt verfügbar und in Python geschrieben. Zudem ist der Code leicht verständlich und erweiterbar [50, S. 1]. Durch die Verwendung dieser Bibliothek und die Vorstellung der Programmiersprache Rust während meines Studiums enstand die Idee diese Arbeit zu schreiben. Einige Lösungen werden deshalb von SymPy inspiriert sein. Der Code soll dabei nicht 1-zu-1 in Rust neu geschrieben werden, sondern mit den Möglichkeiten und Einschränkungen von Rust erweitert werden, auch wenn nur ein Teil des Umfangs von SymPy in dieser Arbeit abgebildet werden kann.

Die Entwicklung von und mit Rust wird immer relevanter. So wird beispielsweise die Entwicklung für den Linux-Kernel immer weiter vorangetrieben [57]. Auch für die Entwicklung unter Android [80] und Windows [53] stehen erste Bestrebungen bereit. Erst letztes Jahr erschienen mit der "Edition 2021" [16] neue Features. Für die nächste Version "Edition 2024" gibt es ebenfalls bereits eine Roadmap [4].

Größere Projekte des wissenschaftlichen Bereichs, in denen Rust explizit verwendet wird, sind leider nicht auffindbar. Es gibt jedoch unabhängige Pakete, die einzelne Gebiete der wissenschaftlichen Programmierung abdecken [13] [12] [59]. Weiterhin können Beiträge gefunden werden, die das Thema aufgreifen und zumeist auf den schweren Einstieg hinweisen [21] [60] [61]. Dies soll in der nächsten Version von Rust verbessert werden [5].

2 Grundlagen CAS

Das folgende Kapitel definiert den Funktionsumfang des entwickelten CAS. Um einen Überblick über die Größe der Arbeit zu geben, soll näher auf SymPy eingegangen sowie aufgezeigt werden, welche weiteren Implementierungen denkbar wären, wenn der Funktionsumfang mit SymPy verglichen wird.

2.1 Grundregeln und Annahmen

Bevor überlegt wird, wie der Aufbau des CAS aussieht und welche Typen benötigt werden, müssen ein einige Vorüberlegungen gemacht werden, welche Grundregeln und Annahmen im System gelten sollen. Ein Teil dieser Regeln stammt aus "SymbolicC++" [82, S. 2] und "Algorithms for Computer Algebra" [24, S. 23 ff.], andere wiederum sind als Einschränkung oder zur besseren Verständlichkeit im Rahmen dieser Arbeit hinzugefügt worden. Es sollen folgende Zahlenräume betrachtet werden:

- $\mathbb{N}$ die Menge der natürlichen Zahlen mit 0, $\{0, 1, 2, 3...\}$
- $\mathbb{Z}$ die Menge der ganzen Zahlen, $\{... - 3, -2, -1, 0, 1, 2, 3...\}$
- $\mathbb{Q}$ die Menge der rationalen Zahlen, $\{... - \frac{2}{1}, -\frac{1}{2}, -\frac{1}{1}0, \frac{1}{1}, \frac{1}{2}, \frac{2}{1}...\}$
- $\mathbb{R}$ 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 verschiedene Vereinfachungen und Bedingungen anzunehmen. So müssen alle Implementierungen nur die Operationen Addieren und Multiplizieren berücksichtigen.

- Jede Substraktion ist grundsätzlich eine Addition. Für Zahlen bedeudet 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, z.B. $2/3 = \frac{2}{3}$ und $-(2/3) = \frac{-2}{3}$. Symbole und Terme werden zu einer Multiplikation vereinfacht, $a/b = b^{(-1)} * a$ 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: $a * (x + y) = a * x + a * y$

2.2 Funktionsumfang

Da wie bereits in der Einleitung gezeigt, ein CAS sowohl einen sehr breiten Umfang haben, als auch spezialisiert sein kann, muss noch definiert werden, welche Möglichkeiten das in dieser Arbeit entwickelte System besitzt. Konkret soll veranschaulicht werden, welche Operationen erlaubt sein sollen und welche Art von Termen verarbeitet werden können:

- Addition von Zahlen (Ganzzahlen, Fließkommazahlen, rationalen Zahlen), z.B. $1+2.1 = 3.1$ oder $\frac{1}{3} + \frac{1}{3} = \frac{2}{3}$

- 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 die Zahlen, Symbolen und Terme in Basis oder Exponent haben. Dabei sollen Potenzregeln in Hinblick auf Genauigkeit des Ergebnisses ausgeführt werden, 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. 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 → Funktionen → Potenzen → Multiplikationen → Additionen

Darüber hinaus sollen Terme soweit wie möglich mit bereits bestehenden Sprachmitteln erweitert und verglichen werden können. Hierfür werden einige Operatorüberladungen implementiert. Bei der Auswertung der Terme soll die Möglichkeit bestehen eventuell ungenaue numerische Ergebnisse zu berechnen. Funktionen und Konstanten sollen später einfach hinzugefügt werden können.

2.3 Überblick SymPy

Das SymPy Projekt beschreibt sich selbst als eine Python-Bibliothek für symbolische Mathematik, die das Ziel hat, ein CAS mit “vollen Funktionsumfang” zu werden [81]. Im Gegensatz zu dem hier begrenzten System beherrscht SymPy z.B. das Handling von Matrizen [50, S. 11] oder auch physikalische Berechnungen [50, S. 14 ff.]. Darüber hinaus können auch 2D- und 3D-Grafiken erstellt werden uvm. [50, S. 4 ff.]. Das Projekt existiert seit mindestens 2007 [73] und wird seither stetig weiterentwickelt.

Verwendung findet SymPy in anderen Projekten, die teils wiederum spezifisch sind, z.B. “ChemPy” [66] oder mehrere Abhängigkeiten vereinen, z.B. “sagemath” [87].

3 Vorstellung der Programmiersprache Rust

In diesem Kapitel sollen die Grundlagen von Rust erläutert werden. Dabei sollen sowohl die Syntax als auch die grundlegenden Konzepte der Sprache und deren Vorteile vermittelt werden. Einige Stellen zur Vorstellung der Sprache sind meiner Seminararbeit "Die Programmiersprache Rust" entommen [32]. Wo nötig, wurden Texte erneuert, erweitert und Quellen aktualisiert.

3.1 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>,
}
```

In diesem 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 [37]. 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 [19]. 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 wurde. Der Zugriff auf den entsprechenden Speicherbereich im Hauptprogramm ist 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 jedoch nur, wenn der Variablen im Hauptprogramm der Rückgabewert zugewiesen wird. Hierzu muss die Variable `matrix` explizit als veränderbar mit dem Schlüsselwort `mut` definiert werden. Der Nachteil dieser Methode ist, dass nicht klar ist, ob es sich um die originale Matrix handelt, oder um 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 lediglich eine Referenz auf die Matrix. Dies ist durch das &-Zeichen gekennzeichnet. In Rust wird dieses Prinzip als "Borrowing" bezeichnet [40]. 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 [48].

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 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 ebenfalls wieder nur der Zeiger freigegeben [37].

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);
```

Die beiden letzten Funktionen können auch einen Rückgabewert besitzen, der unabhängig von der übergebenen Matrix ist. 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 Messge Passing [34] und threadsichere Referenzzähler [6] an. Die Regeln zu Ownership und Borrowing werden noch genauer vorgestellt.

3.2 Was ist Rust und warum sollte es verwendet werden?

Rust ist eine statisch typisierte Programmiersprache, welche vor allem mit den Eigenschaften Performance, Verlässlichkeit und Produktivität eine Alternative zu anderen Sprachen anbieten soll [83]. Die Version 1.0 wurde 2015 veröffentlich. Seither gibt es mit der "Rust 2018 Edition" und "Rust 2021 Edition" eine stetige Weiterentwicklung [29].

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 weitere Aspkekt ist die Implementierung von Features in Rust. So wird beispielsweise Monomorphisierung zur Compilezeit von Generics eingesetzt (vgl. [65, S. 196 ff.]). 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 [58].

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 unter 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 Dokumentationen und das Ausführen von Tests und Benchmarks.

Wo nötig, setzt Rust auf etablierte Lösungen. So wird LLVM genutzt um ausführbare Programme zu erzeugen (vgl. [31] und [30]).

3.3 Grundlegendes

Expression und Statements Rust unterscheided 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 (vgl. [3, S. 122 ff.]).

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 Felder und Feldtypen an. Methoden und Funktionen werden erst nach der Deklaration hinzugefügt, wobei mehrere Implementierungsblöcke möglich sind. Die Implementierungen können Methoden inklusive 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. [3, 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_smth(&self) {}
}
```

trait In Rust gibt es weder Vererbung durch `struct`s, noch klassiche Interfaces. Mit `trait`s kann aber ein ähnliches Verhalten nachgebildet werden. Bei der Definition des `trait` findet die Deklarierung von Methoden und Funktionen statt. Anschließend können `struct`s und `enum`s ein oder mehrere `trait`s 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.

Codebeispiel 9: match

```
let mut var = 19;

let match_var = 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"),
};
```

3.4 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 für `struct`s können Implementierungsblöcke definiert und Traits implementiert werden. Mit dem bereits vorgestellten `match` kann geprüft werden, welche Ausprägung das `enum` besitzt. 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 Fuktionen und Methoden. Außerdem wird ein `trait` implementiert. Die Syntax ist dabei identisch zu der Implementierung von `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 { ... }
```

3.5 Generics

Rust bietet ebenfalls generische Typparameter an. Diese können bei `struct`, `enum`, `trait` und Funktionen verwendet werden. Um eine hohe Performance zu gewährleisten, wird Mono-

morphisierung zur Compilezeit eingesetzt (vgl. [65, 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: Veranschaulichung der Monomorphisierung [39]

```
// 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 werden dazu 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 kann.

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 [46]

```
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 Typparemeter und/oder Einschränkungen
struct ScreenGeneric<T: Draw> {
    components: Vec<T>,
}
// Oder falls mehrere Typparemeter 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
    ],
};
```

3.6 Operatorenüberladung

In Rust können Operatoren überladen werden und für eigene als auch bestehende Typen implementiert und erweitert werden [18] [3, S. 246 ff.]. Diese Funktionlität kann im CAS dazu genutzt werden, um z.B. einen bestehenden Term ohne Funktionsaufruf zu verändern. Stattdessen können gewohnte Sprachmittel verwendet werden. Weiterhin 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 { ... }
```

3.7 Ownership, Borrowing und Lifetimes

Im Folgenden sollen die Konzepte Ownership, Borrowing und Lifetimes vorgestellt werden. Diese tragen wesentlich zur erwähnten Speichersicherheit bei. Der Compiler beachtet die folgenden Regeln [38] [41].

- 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 [43]. 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 nicht ständig den Owner der Daten ändern zu müssen, z.B. bei einem Funktionsaufruf, werden mit dem Borrowing Referenzen erstellt. Hierauf beziehen sich die beiden letzten Regeln. Hierbei gilt außerdem, dass es nur veränderbare Referenzen geben kann, wenn der Owner veränderbar ist. Falls es eine veränderbare Referenz gibt, kann der Wert 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 enstpricht dabei den Codeblöcken, in denen die Variablen definiert sind. Die inneren Lifetimes im Beispiel könnten jedoch 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 im Gegensatz dazu gültig, weil der eigentliche Wert erst nach der Referenz freigegeben wird.

Codebeispiel 16: Lifetime Veranschaulichung [35]

```
{
    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 Deklarierung der Lifetimes ist nötig, wenn beispielsweise eine Funktion mehr als eine Referenz entgegennimmt und wieder zurückgibt. Dies soll am folgendem Beispiel verdeutlicht werden. Die Lifetimedeklarierung `'a` gibt an, dass sowohl die beiden Auswahlmöglichkeiten als auch der Rückgabewert diesselbe 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 nicht mit der Lifetime des Rückgabewerts überein.

Codebeispiel 17: Lifetime Beispiele [47]

```
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
    }
}
```

3.8 Stack und Heap

Beide Speicherbereiche müssen in Rust seperat betrachtet werden. Datentypen, die eine bekannte Größe haben, wie z.B. eine Ganzzahl, werden in Rust auf dem Stack nach dem Prinzip "last in, first out" abgelegt. Hierdurch hat der Stack eine bekannte Größe. Durch diese Einschränkung kann der Zugriff auf den Stack zügig erfolgen. Typen deren Größe zur Compilezeit nicht bekannt sind, z.B. ein Vektor, müssen in Rust auf dem Heap abgelegt werden. Da für diese Daten erst ein passender Bereich in der benötigten Größe gesucht werden muss, ist der schreibende Zugriff langsamer auf dem Stack. Auch der lesende Zugriff erfolgt über eine Referenz, sodass dieser mehr Zeit in Anspruch nimmt [42] [65, S. 233 ff.].

Besonders wichtig ist dieser Unterschied, wenn eine Variable freigegeben wird. Liegt der Wert auf dem Stack, muss nur dieser freigegeben werden. Liegt der Wert hingegen auf dem Heap, muss sowohl der Heap, als auch die Referenz auf dem Stack freigegeben werden.

3.9 Copy und Clone

Rust bietet zwei Möglichkeiten Werte zu duplizieren. `Copy` ist dabei möglich, wenn der Wert nur auf dem Stack liegt. Die Zuweisung "a = b;" kopiert den Wert b. Sowohl "a" als auch "b" haben ein eigenes Ownership. Die Funktionsweise ähnelt `memcpy` in C [65, S. 259 ff.].

Liegen die Werte hingegen auf dem Heap, muss die `clone`-Methode für den Typen implementiert werden. Hierdurch ist die Kopie explizit und kopiert standardmäßig ebenfalls die Daten auf dem Heap. Durch das Anpassen der Methode können z.B. auch nur die Stackwerte kopiert und ein Referenzzähler implementiert werden. Die `clone`-Methode erlaubt somit flexible Strukturen [65, S. 260 ff.]. Bei einer Zuweisung wird ansonsten wie bereits gezeigt das Ownership abgegeben.

4 Implementierung des CAS

Das folgende Kapitel stellt die Implementierung des CAS in den Vordergrund. Hierzu sollen auch Überlegungen zu früh auftretenden Problemen gemacht werden.

4.1 Überlauf und Ungenauigkeit

Ein bereits zu Beginn auftretendes Problem für das CAS ist der Umang von Ganzzahlüberläufen und die Ungenaugikeiten der Rechnung mit Fließkommazahlen [33]. 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
0.30000000000000004
thread main panicked...
```

Ein Blick zu SymPy zeigt, dass Lösungen direkt aus Python verwendet werden können, die in Rust nicht existieren. So besitzen Ganzzahltypen bereits eine beliebige Präzision [14]. Für Fließkommazahlen ist die Klasse `Float` implementiert, die eine ungebrenzte Genauigkeit erlaubt. Dabei kann mit dem zweiten Parameter die eigentliche Präzision bestimmt werden. Es wird darüber hinaus angeraten den eigentlichen Wert als String zu übergeben, da der native `float`-Typ in Python eine Ungenauigkeit hervorrufen kann [67].

Codebeispiel 19: SymPy Präzision

```
x = 123456789012345678901...
print(type(x))
// <class int>

print(sympy.Float('0.1', 100) + sympy.Float('0.2', 100))
// 0.3000000000000000000000000000000000000000000000000...

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

In “Modern Computer Algebra” [23, S. 29 ff.] und “Computer Algebra - Concepts and Techniques” [49, S. 81 ff.] sind theoretische Implementierungen für Ganzzahlwerte angegeben. Hiervon könnte in dieser Arbeit Gebrauch gemacht werden und diese Typen selbst implementiert und für Fließkommazahlen erweitert werden. Allerdings existieren bereits die beiden Crates (Abhängigkeiten im Rust-Ökosystem) “num” [56] und “bigdecimal” [1], welche Typen mit ungebrenzter Präszision anbieten.

Folgend soll die Performance zwischen den primitiven Typen auf dem Stack, den primitiven Typen auf dem Heap und den Crates verglichen werden. Wie zu erwarten, ist die Speicherung auf dem Stack am effizientesten und 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 verwendet der Typ `BigInt` einen Vektor. Wie in Kapitel 3.2 bereits erwähnt, wird bei einem Zugriff zusätzlich geprüft, ob der Index gültig ist, dies kostet ebenfalls Zeit.

Im Benchmark werden Zahlen von 10 bis 9999 durchlaufen und in jedem Durchlauf eine Berechnung mit dem vorher existierenden Wert durchgeführt.

Codebeispiel 20: Benchmark Primitive Typen Stack Ganzzahl

```
let mut x: i64 = 0;
for i in 10..10000 {
    x += i;
    x /= 3;
}
assert_eq!(x, 4999);
```

Tabelle 1: Ergebnisse Benchmark Rust-Typen

Test	**Zeit Median in ns**	**Unterschied min. und max. Laufzeit in ns**
Primitiv int	18.505	396
Primitiv int Heap	50.293	1.683
Primitiv float	36.975	809
Primitiv float Heap	74.324	920
num Crate	548.640	42.989
bigdecimal Crate	7.447.000	474.681

Bisher ist es mit Rust-Benchmarks noch nicht möglich den Speicherverbrauch zu messen. Allerdings kann dieser für die gewählten Beispiele auch selbst gut bestimmt werden. Hierbei geht es lediglich um die eigentlichen Werte, mit welchen Berechnungen durchgeführt werden.

Primitive Typen Stack Die Datentypen `i64` und `f64` verbrauchen 8 Bytes. Es werden somit nie mehr als $8 * 4 = 32$ 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`) durch die Addition mit x (bzw. dem TypeCasting) behandelt und verbraucht ebenfalls 8 Byte. Bei der Addition $x + i$ wird, wie bereits erwähnt, bei einem Funktionsaufruf eine Kopie der Werte erstellt, die den Copy-Trait implementieren. Die Typen `i64` und `f64` implementieren den Copy-Trait [76] und somit wird sowohl von x als auch von i zur Addition eine Kopie erzeugt.

Primitive Typen Heap Der Speicherverbrauch der eigentlichen Werte ist wie bei den primitiven Typen auf dem Stack. Davon entfallen 8 Bytes auf dem Heap statt auf dem Stack. Zur Berechnung müssen die `Box`-Werte dereferenziert werden. Zusätzlich befindet sich ein Zeiger auf dem Stack durch `Box`. Die Größe hängt von der verwendeten Architektur ab [45].

num und bigdecimal Der Datentyp BigInt verwendet einerseits ein Vorzeichen, welches 1 Byte verbraucht [20], als auch den Datentyp BigUint, der wiederum einen Vektor mit BigDigit erhält. Das Vorzeichen ist als `Enum` umgesetzt [55]. Betrachtet man diese Definition genauer, 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 jedoch

dieser Verbrauch, da sich der Vektor vergrößert. Zudem liegen nicht mehr alle Daten auf dem Stack, sondern alle BigDigits auf dem Heap, was die Zugriffsgeschwindigkeit zusätzlich erhöht.

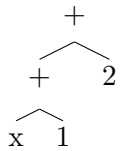
Der Typ bigdecimal verwendet laut Beschreibung den Typ BigInt und zusätzlich einen 64-Bit-Integer um die Position der Dezimalstelle zu bestimmen [2]. Dadurch erhöht sich der Speicherverbrauch nur unwesentlich.

Entscheidung der verwendeten Typen Im Rahmen dieser Arbeit ist eine eigene Implementierung der Typen für beliebige 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 wiederrum die Implementierung mit primitiven Typen von Vorteil.

Da Rust Generics untersützt, 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 sowohl eine spätere Erweiterung der Crate-Typen als auch die Entwicklung der eigenen Typen.

4.2 Grundlegende Datenstruktur

Ein erster Ansatz die Datenstrukturen darzustellen war ein binärer Baum, dessen Knoten die Operanden und die Blätter einzelne Werte sind. Dies entspricht dem abstrakten Syntaxbaum nach dem Parsen. Für den Term $x + 1 + 2$ ergibt sich folgender Baum.



In Rust sieht die Implementierung folgendermaßen aus. Die Ausprägung Add benötigt für die Werte immer die Struktur `Box`, die die Daten auf den Heap speichert. Zur Compilezeit lässt sich die Größe auf dem Stack nicht bestimmen, da unklar ist, wie viele Ebenen Add beinhaltet [36].

Codebeispiel 21: 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 durch das Ownership ein Problem. Man benötigt auf jeden Knoten und Blatt eine veränderbare Referenz. In diesem Beispiel würden im einfachsten Fall das untere `BTree::Add` durch `BTree::Symbol` ersetzt und das rechte `BTree::Number` aktualisiert 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 in Anspruch 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 weiterhin an `parent_1` übergeben werden.

Codebeispiel 22: BTree Referenzen

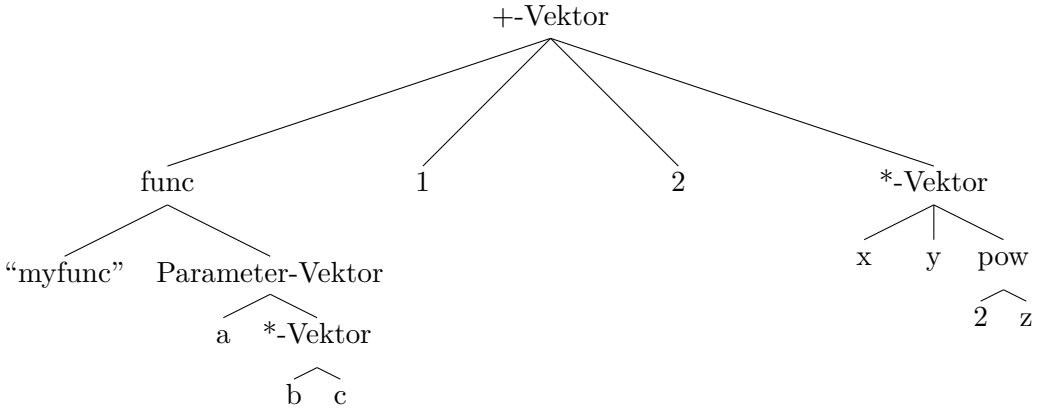
```
let mut leave_1 = BTree::Number(1);
let mut leave_2 = BTree::Number(2);
let mut parent_1 = BTree::Add(Box::new(leave_1), Box::new(leave_2));

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

Hier könnte nun versucht werden, das Problem mit Referenzzählern (vgl. [78] und [65, S. 293 ff.]) zu lösen, die dafür gedacht sind, mehrere Referenzen zu erzeugen. In Kombination mit anderen Strukturen in Rust können diese Referenzen auch veränderbar sein (vgl. [77], [79] und [65, S. 299 ff.]).

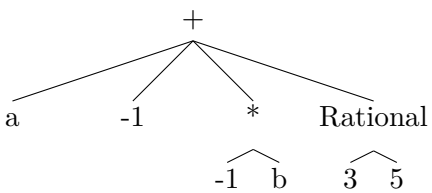
Ebenfalls möglich wäre die Nutzung von Rohzeigern in Verbindung mit `unsafe`, was Code in Rust kennzeichnet, der auf undefinierten Speicherbereich zeigen kann [44].

Da beide Varianten aber sich entweder auf den Codeumfang und -verständlichkeit auswirken oder der Hauptgrund Rust zu benutzen umgangen wird, soll, wenn möglich auf diese Varianten verzichtet werden. Damit ausgeschlossen ist aber auch die Möglichkeit den Baum in situ zu verändern

Eine Alternative wäre lediglich 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, könnten diese auch direkt in einem Vektor gespeichert werden. Für die Addition und Multiplikation würde dann der Baum für den Term $myfunc(a, b * c) + 1 + 2 + x * y * 2^z$ beispielsweise wie folgt aussehen:



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



Eine ähnliche Struktur findet sich ebenfalls in “SymbolicC++” [82, S. 284 ff.]. SymPy verwendet ebenfalls diese Struktur [50, S. 18 ff.] [15]. Da sich dieser Aufbau anscheinend für

ein CAS eignet, soll dieser auch in diesem System verwendet werden. Hierrauf bauen alle weiteren Operationen auf, wie z.B. das Zusammenfassen des Terms oder Substitution.

4.3 Grundtypen des CAS

In diesem Kapitel soll definiert werden, welche `structs`, `traits` oder `enums` benötigt werden, um das CAS zu erstellen. Darüber hinaus soll grob umrissen werden, welche Implementierungen im weiteren Verlauf sinnvoll sein können und welchen Zweck diese erfüllen. Weiterhin ist noch zu erwähnen, dass hier nicht definiert werden soll, welche `Derive macros` diese Typen erhalten. 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 welchem gerechnet werden kann und nicht ständig manuell Type Castings durchgeführt werden müssen. So ist es 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` implementiert werden kann. Auch hier kann der Typ direkt die Rechnung übernehmen. Hierzu werden die Operatoren `std::ops::Add` und `std::ops::Mul` für das 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 der vorliegenden 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 eine rationale Zahl erzeugt wird. Zudem wäre 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 nutzbar sein. In diesem Fall sind dies die beiden `enums`, welche gerade erwähnt worden sind. Zudem werden Funktionen als Bedingung definiert, um die neutralen Elemente der Addition und der Multiplikation zurückzugeben sowie die 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 besitzen 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, wird ein Parser benötigt, der die generierten Tokens oder die entstehende Datenstruktur in die Struktur des Systems überführt.

4.4 Datenstruktur in Rust

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

Codebeispiel 23: 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. Hierbei ist die Nutzung von 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 darüber hinaus 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 und die Parameter einfach übergeben werden.

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 einerseits die Ausführungsreihenfolge beachtet werden, andererseits soll diese Methode leicht erweiterbar sein.

Im vorliegendem System wird die Methode rekursiv implementiert. Eine iterative Lösung wäre ebenfalls möglich. Aufgrund des Ownership und Borrowing in Rust wäre solch eine Lösung aber komplex, weniger leserlich und unverständlich [22].

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 erfolgt folgendermaßen. Die beiden zusätzlichen Parameter `evaler` und `hard_eval` werden noch näher in den Kapiteln zu "Erweiterbarkeit in Rust und Beispiele" und "Ungenauigkeit in Kauf nehmen" näher erläuert.

Bei der Entscheidung, ob das Element direkt bearbeitet wird oder ein neuer `Ast` zurückgegeben wird, wurde der Definition von SymPy gefolgt, sodass der originale Term nicht verändert sondern ein neuer erstellt wird.

> "Another important property of SymPy expressions is that they are immutable."
> [50, S. 19]

Mit den bereits genannten Einschränkungen, ergibt sich, dass die `clone`-Methode aufgerufen werden muss, da einige Daten auf dem Heap abgelegt werden. Bei einer Zuweisung oder einem Rückgabewert würde ansonsten der originale `Ast` das Ownership abgeben. Der `Copy`-Trait kann für `Ast` nicht implementiert werden, da bereits Vektoren diesen Trait nicht implementieren.

Codebeispiel 24: 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 Nummerntyp darstellen. Dieser wird aber noch nicht eingeschränkt, da sonst diese Einschränkungen später für alle implementierenden Blöcke gelten müssen. Dies kann z.B. nicht gewünscht sein, wenn eine grundlegend andere Art implementiert werden soll, bei der der Trait `NumberType` keine Verwendung finden soll um die Terme zu verarbeiten. Mit diesem Ansatz kann auch die definierte Ausführungsreihenfolge der Ausprägungen bestimmt werden. 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 werden nahezu alle 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 im Kapitel 3.5 zu Trait Boundaries gezeigt, aus Gründen der ungewissen Speicheranforderung auf dem Heap ab. 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 [7].

Codebeispiel 25: Term 1 Stack und Heap

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

Codebeispiel 26: 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, es 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     |
                                    +--------+--------+
```

5 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.

5.1 Prüfung von bestehenden Bibliotheken zum Parsen

mexprp Rust [52] Diese Bibliothek wurde geprüft, 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 die gewünschte Form zu bringen, da das Enum `Term` einen Abstract Syntax Tree implementiert wie anfänglich angedacht. Über diesen müsste nur einmal iteriert werden, um die modifizierte Version zu erhalten.

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 Genaugikeit verwendet, sondern num und bigdecimal.

meval Rust [51] Diese Bibliothek erscheint recht unkompliziert und einfach in der Nutzung. Die geparsten Tokens werden als UPN (umgekehrte polnische Notation) ausgewertet. Dies könnte man prinzipiell dazu nutzen, den modifizierten Ast zu erzeugen. Ein kleiner Nachteil ist, dass nur Fließkommazahlen ausgewertet werden. Sollte der Lexer erweitert werden, sodass eine beliebige Genauigkeit erreicht werden soll, müsste dieser Part neu geschrieben werden. Zudem wurde diese Bibliothek nicht bei der initialen Recherche in Betracht gezogen und nicht weiter überprüft. Während der Entwicklung des CAS wurde interessehalber vertiefend reingesehen und die Enums `Operator` und `Token` größtenteils übernommen. Wäre der Lexer nicht bereits zu einem großen Teil implementiert, würde die Möglichkeit genutzt, die Tokens aus dieser Bibliothek zu erzeugen.

evalexpr Rust [17] Diese Bibliothek erscheint sehr umfangreich. Allerdings lässt sich fast nur die `eval`-Funktion nutzen, da fast alle anderen Strukturen nicht mit dem Schlüsselwort `pub` versehen sind. Es war daher nicht möglich, den Tokenizer bzw. den Baum direkt zu nutzen, in dem sich ein bereits geparster String befindet. Daher kam eine Nutzung dieser Bibliothek nicht in Betracht.

tinyexpr C [9] 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 zu komplex wird. Hierbei wurde tinyexpr näher betrachtet. Allerdings müssen Variablen bereits vor der lexikalischen Analyse zumindest bekannt sein [8]. Da sich generell gegen diesen Schritt entschieden wurde, wurde auch diese Bibliothek nicht verwendet.

Fazit Eine geeignete Bibliothek zu finden, die den Anwendungszweck erfüllt und dabei nicht zu weit weg vom eigentlich Ziel ist, war nicht möglich. Daher wurde sowohl die lexikalische Analyse als auch das Parsing selbst implementiert. Die wichtigsten Eigenschaften hierzu wurden aus dem Kurs 01810 Übersetzerbau herausgearbeitet, den ich im Wintersemester 17/18 belegte.

5.2 Implementierung der lexikalischen Analyse bzw. Tokenizer

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

Die Tokens werden dabei als Enum implementiert. In diesem Abschnitt wird auf die Implementierung von Generics verzichtet und nur der Typ `PrimNum` erlaubt.

Im Enum werden folgende Werte benötigt:

Codebeispiel 27: 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, Terme zu gruppieren und die eigentliche Operatorenreihenfolge zu verändern, wie es in der Mathematik üblich ist. Es werden die Zeichen "(" und ")" erkannt.

Comma Dient als Seperator von Funktionswerten.

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 Ganz- oder Fließkommazahl. Eine Fließkommazahl wird daran erkannt, dass das erste Zeichen ein "." ist oder die gesamte geparste Zeichenkette einen "." enthält. Es sind nur Zahlen erlaubt. Die E-Schreibweise $1.2E10$ wird nicht unterstützt.

5.3 Implementierung Parser

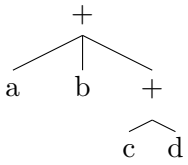
Als Parser kommt eine einfache Implementierung der Operator-Vorranganalyse in Frage, wie sie in Kurs 1810 Kurseinheit 3 Kapitel 3.3.2 beschrieben ist [28, S. 83 ff.]. Es handelt sich somit um eine Bottom-Up-Analyse. Dabei wird der Algorithmus so implementiert, dass der momentan gültige Operator in einer `while`-Schleife läuft, um einen Vektor befüllen zu können. Das erste Element des Vektors wird hierbei jeweils vor der Schleife geparst. Die restlichen

Elemente in der Schleife selbst. Darüber hinaus werden Operatorenüberladungen des `Ast`-Enum verwendet.

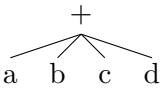
Es handelt sich um einen LR(1)-Parser, da der Parser ein Zeichen vorrausschauen kann, um die nächste Aktion zu bestimmen. Der Stack ist im weitesten Sinne bereits ein gültiges `Ast`-Element, das befüllt wird. Die `shift`-Funktion kann als Voranschreiten im Iterator angesehen werden. Die `reduce`-Funktion entpricht dem Anwenden der Operatorenüberladungen auf dem gültigen `Ast`-Element.

Das Parsing von Funktionsargumenten beginnt wieder auf der untersten Stufe zum parsen von Additionen. Auch werden bereits Variablen und Konstanten unterschieden. So enthält der Ast für den $a + \pi$ sowohl “Ast::Symbol(“a”)” als auch “Ast::Const(“π”)”.

Darüber hinaus vereinfachen die Operatorenüberladungen den Baum wie angedacht. So wird der Term $a + b + (c + d)$ nicht wie der folgende Baum geparst:



Es entsteht vielmehr sofort der Baum:



6 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.

6.1 Änderung des Terms mithilfe von Operatorenüberladung

Der Trait `NumberType` soll bereits die meisten nötigen Operatorenüberladungen von den zu implementierenden Typen verlangen. Die wichtigsten Operationen sind `Add` und `Mul`, da diese vom CAS benutzt werden. Operatoren, die ebenfalls interessant im weiteren Verlauf sein können, sind der Modulo-Operator `Rem` (z.B. 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 28: 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 { ... }
```

Ebenfalls sollen für das Enum `Ast` Operatoren überladen werden. Hierbei wird sich auf die Operatoren "Add" und "Mul" beschränkt. Mit dem zusätzlichen Generic-Typ ergibt sich die Defintion:

Codebeispiel 29: 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,
{ ... }
```

Falls möglich sollen dabei die Überladungen für `Ast` Zusammenfassungen ausführen. Das bedeutet, es werden konkret für Addition und entsprechend für Multiplikation die folgenden Vereinfachungen vorgenommen.

Codebeispiel 30: Operatenüberladung für Addition

```
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]
    )
```

6.2 Addieren und Multiplizieren (Standardauswertung von Termen)

Die prinzipielle Vorgehensweise um Additionen und Multiplikationen auszuführen, orientiert sich an dem Vorgehen aus SymPy (vgl. [68] und [69]). Beide Verfahren sind im Grundsatz identisch. Es ändert sich hierbei 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 ebenfalls im System implementiert. Kernstück ist eine `HashMap` deren Schlüssel und Werte vom Typ `Ast<N>` sind. Der Schlüssel ist hierbei der eigentliche Term, der in der Addition auftritt. Der Wert ist 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 beim vorherigen Wert wird wieder versucht eine Zahl zu extrahieren. Dies gelingt für $x * y$ nicht. Bei der Multiplikation kann somit das neutrale Element 1 für diesen Subterm hinzugefügt werden. Die HashMap erhält nun den aktualisierten Wert $[x * y] = 3$.
- 3. Wert: Alle Zahlen, also der genutzte Nummerntyp, können zu dem Zahlenergebnis addiert werden. Hierzu wird die bereits vorhandene Operatorenüberladung genutzt.
- 4. Wert: Alle anderen Ausprägungen des `Ast`-Enum (in diesem Fall Symbol) werden ähnlich der Multiplikation behandelt. Jedoch wird hierbei immer der Wert 1 addiert. Die HashMap erhält hier nun den Wert $[a] = 1$.
- 5. Wert: Dieser Wert scheint wieder ein Symbol zu sein. Allerdings wurde bereits erwähnt, dass diese Form intern als $+(-1 * a)$ gespeichert wird. Somit wird der HashMap wieder eine Multiplikation hinzugefügt, die den Multiplikator -1 hat. Folglich wird die HashMap aktualisiert mit $[a] = 0$.

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

Zum Schluss können die Ergebnisse wieder zurück in die Ausprägung `Ast::Add` überführt werden. Das Zahlenergebnis kann hierbei ohne weitere Prüfungen hinzugefügt werden. Bei der HashMap hängt dies vom Wert des Schlüssels ab.

- Wert == 0: Schlüssel wird dem Vektor nicht hinzugefügt
- Wert == 1: Schlüssel wird dem Vektor hinzugefügt
- Alle anderen Werte: Es wird dem Vektor `Ast::Mul(vec![Wert, Schlüssel])` hinzugefügt

Um das Verfahren zu implementieren, müssen für das `Ast`-Enum noch die Traits `Eq`, `PartialEq` und `Hash` implementiert werden. Erst dann kann der Typ `Ast` als Schlüssel in der HashMap verwendet werden. Im den meisten Fällen kann die Implementierung über Makros erfolgen. In diesem Fall ist es aber nur für `PartialEq` möglich. Da z.B. Vektoren die Traits `Eq` und `Hash` nicht implementieren, müssen die Traits manuell implementiert werden. Der `Eq`-Trait ist dabei aber nur deklarativ. Es muss keine Funktion ausgefüllt werden. Dies übernimmt der Compiler [75]. Der `Hash`-Trait kann ebenfalls einfach implementiert werden. Hierbei können alle vorhandenen Werte als Hash zusammenfasst werden.

6.3 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 postiv und negativ. Der Term $1/3 + 1/3$ wird also zu `Num(Rational(2, 3))` ausgewertet. Der Term $1/3 + 1/3 + 1/3$ wird somit zu `Num(Int(1))` ausgewertet.

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 $x^a * x^b = x^{(a+b)}$ 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 im Kapitel zur Addition und Multiplikation vorgestellte Vorgehensweise.

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

Darüber hinaus wird derselbe Algorithmus wie in SymPy implementiert, um zu prüfen, ob eine Wurzel eine perfektes Ergebnis liefert [70]. Hierbei wird das "Newtonverfahren" genutzt, das Näherungswerte für die Nullstullen von nichtlinearen Gleichungssystemen findet [86, S. 28 ff.]. Bei dieser Implementierung wird der Wert näherungsweise bestimmt und danach überprüft, ob die Potenz aus dem Wert und der n-ten Wurzel den gesuchten Wert abbildet. In diesem Fall kann das entsprechende Element vereinfacht werden.

6.4 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 31: Defintion 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 bei EvalFn angegeben werden. Die Felder sind teils Vektoren, teils HashMaps, die entsprechende Funktionen zur Vereinfachung vorhalten. Der Parameter mit dem Typ `&mut HashMap<Ast<N>, Ast<N»` wurde bereits genauer vorgestellt.

adders Funktionen, die den Typ `Ast::Add` weiter vereinfachen können. So ist es möglich, entsprechend die Einträge der `HashMap` zu überprüfen und zu verändern. Der zweite Parameter vom Typ `bool` gibt an, ob es sich um eine explizite Berechnung handelt. Dies wird relevant, wenn auch ungenaue Werte berechnet werden sollen.

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 kein Beispielterm gefunden werden 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 darüber hinaus einen Rückgabewert, da der entsprechende `Ast` nicht direkt bearbeitet wird, da sich dieser 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 $sqrt(4)$ kann zu 2 vereinfacht werden. Dabei bleiben Funktionen, die `EvalFn` nicht bekannt sind im Term unverändert erhalten. Falls eine Auswertung nicht möglich ist, ist dies ebenfalls der Fall. Ansonsten besteht für die Funktion immer die Möglichkeit den Term in ein anderes `Ast`-Objekt umzuwandeln. Dieses Feld ist als HashMap implementiert, da eine Funktion nur eine Auswertung besitzt.

consts Funktionen die nur ein `Ast` zurückgeben. Eine Einschränkung, dass dies ein Zahlwert sein soll, ist nicht möglich. Dieses Feld ist ebenso als `HashMap` umgesetzt.

6.5 Implementierung von Substitution

Um einen Term mit Variablen auszuwerten, müssen diese im `Ast`-Enum substituiert werden können. Hierzu wird die bestehende `eval`-Methode erweitert, sodass 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 entsprechende `Ast`, mit dem substituiert werden soll.

Codebeispiel 32: 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(),
        }
    }
```

Zu überlegen wäre noch, ob der übergebene Term vor der Substituierung ebenfalls evaluiert wird. In der momentanten Implementierung wird jedoch ein bereits evaluierter Term erwartet um diesen nicht doppelt zu evaluieren.

6.6 Implementierung von expand

Unter Umständen ist es interessant einen Term vereinfacht vorliegen zu haben. Die Funktion "expand" kann z.B. bei der Vereinfachung des Terms $(x+1)*(x-2)-(x-1)*x$ hilfreich sein [84]. Wird der Term ausmultipliziert, kann dieser anschließend auf -2 vereinfacht werden. Im Folgenden sollen die folgenden Terme auf dieser Art vereinfacht werden können.

Multiplikation von Additionen bzw. einzelnen Elementen 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.

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 zusätzlich über alle Elemente im Vektor von `Ast::Add` iteriert werden, um das derzeitige Element zu bestimmen. Die Veränderung kann 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 Darüber hinaus soll es möglich sein, für mathematische Funktionen die `expand`-Methode aufrufen zu können. Hierzu wird das struct `EvalFn` erweitert.

Codebeispiel 33: EvalFn nach expand

```
pub struct EvalFn<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 $log(x^z * y)$ der Term $z * log(x) + log(y)$ werden. Zuordnungen zu Funktionen finden erneut über eine HashMap statt.

6.7 expand Implementierungshinweise

Bei der Implementierung dieser Methode sind 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 veränderbare Referenz auf das eigene Objekt und geben auch solch eine wieder zurück. Die Implementierung von

`expand` gibt allerdings eine eigenständige Instanz von `Ast` zurück. Das Ownership wird hierbei abgegeben. 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. Entweder kann das zurückgegebene Objekt mit `.clone()` geklont und als eigenständiges Objekt zurückgeben werden oder es werden die Funktionen auf dem Objekt aufgerufen, um danach das Objekt als solches zurückgegeben.

Codebeispiel 34: 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 das 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 nach den oben genannten Regeln ein neues Element hinzugefügt. Der Exponent bleibt jedoch in jedem Durchlauf dersselbe. Da `Ast` weiterhin nicht den Copy-Trait implementiert und die Initialisierung von Box das Ownership des Wertes bekommt, muss hier `.clone()` ausgeführt werden.

6.8 Implementierung von simplify

Um andere Arten von Vereinfachungen vorzunehmen, wird die Methode “simplify” in Anlehnung von SymPy [50, S. 7 ff.] erstellt. Hier werden verschiedene Möglichkeiten geprüft einen Term effizient zu vereinfachen.

In dieser Arbeit sollen folgende Terme vereinfacht werden können:

1. $a^x * b^x$ wird zu $(a * b)^x$.
2. $(x^3+x^2+x+1)/y/x*(y^2+y)/z$ wird zu $(x^2+x+1+\frac{1}{x})*(y+1)/z$ Wichtig ist hierbei, dass die Divisoren richtig zugeordnet werden und eine möglichst große Vereinfachung bewirken.
3. Als Beispiel soll wieder die log-Funktion als Erweiterung implementiert werden. So wird aus $2 * log(x) + log(b)$ der Term $log(x^2 * b)$.

Die zentrale Frage besteht hier darin, wie die Komplexität eines Terms ermittelt wird. Die Standardimplementierung von SymPy zählt hierzu im Term die nötigen Operationen. So erhält der Term $(x^2 + x + 1)/y$ einen Wert von 4. In dieser Arbeit soll die Komplexität ebenfalls so ermittelt werden. Hierzu wird die Methode `count_ops` implementiert.

Da sich die ersten beiden Regeln zum Vereinfachen überschneiden können, soll darüber hinaus angegeben werden, welche Art der Vereinfachung durchgeführt werden soll.

Regel 1 Die am einfachsten zu implementierende Möglichkeit der Vereinfachung ist die erste Regel. Hierzu werden in einem `Ast::Mul` alle Werte iteriert und wie bereits in der Standardauswertung eine `HashMap` befüllt. Diesmal ist der Schlüssel allerdings der Exponent und der eigentliche Wert die Basis. Zum Schluss wird für jeden Eintrag entsprechend ein neues `Ast::Pow`-Element erzeugt und dem Ergebnis hinzugefügt.

Regel 2 Bei der zweiten Regel ist die Zuordnung der Divisoren zu den Additionstermen eine Schwierigkeit, die gelöst werden muss. Schlussendlich wurde eine Lösung gewählt, die jedem Divisor einem Dividenden zuordnet, die die größte Einsparung erbringt. Diese Kombination wird durchgeführt. Die Zuordnung läuft so ab, dass alle möglichen Divisoren für alle möglichen Dividenden geprüft werden. Hierbei wird die Komplexität des Terms vor und nach Durchführung der Division gemessen und verglichen.

SymPy löst dies anders. Der Term $(x^3 + x^2 + x + 1)/x * (x^2 + x)$ wird hierbei ausmultipliziert und dann ein größter gemeinsamer Teiler gesucht (vgl. [72] Aufrufe `cancel`, `_mexpand`, `together`). Einerseits erscheint diese Möglichkeit sehr mächtig, andererseits komplex und fehleranfällig zu implementieren.

Um die bisher gesammelte Erfahrung mit Rust in diese Arbeit zu bringen, wurde eine eigene Implementierung gewählt.

Regel 3 Die letzte Regel soll es erlauben, benutzerdefinierte Regeln zur Zusammenfassung hinzuzufügen. Hierzu wird wieder das Struct `EvalFn` erweitert, um Funktionen hinzufügen zu können, die nacheinander aufgerufen werden. Als Beispiel dient hierbei die `log`-Funktion, die nun das Gegenteil von "expand" bewirkt. Eine Besonderheit hierbei ist, dass diese Funktion sowohl in Multiplikationstermen als auch in Additionstermen aufgerufen werden muss.

6.9 simplify Implementierungshinweise

Gerade bei der Implementierung der zweiten Regel ist der Aspekt "Borrowing" hervorgetreten. Die ursprüngliche Idee ist, im Term alle Summen und alle anderen Terme zu trennen. Hiernach sollte über alle Summen iteriert werden und der richtige Divisor gefunden werden. Sollte sich bei einer späteren Summe herausstellen, dass der Divisor eine größere Einsparung bringt, soll die vorherige Summe noch einmal mit anderen Divisoren überprüft werden. Allerdings ist es durch das "Borrowing" nicht möglich, einem Vektor ein Element hinzuzufügen, über den gerade iteriert wird.

Die Lösung wird dabei folgendermaßen implementiert. Durch ein zweites Array wird gespeichert, ob es noch ein Element gibt, das überprüft werden muss. Dies geschieht durch einen "bool"-Wert am entsprechenden Index. Nach der Iteration muss der entsprechende Index im Hilfsarray auf "true" gesetzt werden. Im Falle, dass der Divisor wechselt, muss der entsprechende Index auf "false" gesetzt werden. Die Schleife läuft dann solange wie ein "false"-Wert im Hilfsarray vorhanden ist.

7 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.

7.1 Terme mit Fließkommazahlen berechnen und Ungenauigkeit in Kauf nehmen

Um ein einzelnes numerisches Ergebnis zu erhalten, sofern alle Variablen im Term substitutiert 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 Ungenaugkeit verbunden sein.

Der Parameter `hard_eval` soll dabei allen berechnenden Funktionen, insbesondere auch später hinzugefügten 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 dem Fall von `PrimNum` werden 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. Der Aufruf von `hard_eval` gibt dagegen den Fließkommawert `1.4142135623730951` zurück, wodurch bereits eine Ungenauigkeit entsteht.

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 erläutert ist. Bei der Ausführung mit `hard_eval` wird hingegen das Ergebnis 2.0000000000000004 zurückgegeben. Dies hat den Grund, da zuerst Potenzen ausgeführt werden und danach Multiplikationen durchgeführt werden. Andere Nummerntypen, die hier eine höhere Genauigkeit erlauben bzw. solche Sonderfälle abfangen, können hierbei natürlich auch immer noch ein genaueres Ergebnis liefern.

7.2 Implementierung der limit-Funktion

Ursprünglich war geplant, dass die "limit"-Funktion als Beispiel implementiert wird, wie dynamisch Funktionen dem CAS hinzugefügt werden können. Da dies aber tatsächlich bereits z.B. mit der Sinusfunktion geschehen ist, kann sich auf die eigentliche Implementierung konzentriert werden. In dieser Arbeit werden nur einige Heuristiken der "limit"-Funktion implementiert [89].

Die Arbeit von Dominik Gruntz "On Computing Limits in a Symbolic Manipulation System" [27] stellt darüber hinaus noch weitere Methoden vor, wie Limits berechnet werden können und besitzt einen eigenen Algorithmus, der neben Heuristiken in SymPy verwendet wird [71]. Die Implementierung dieser Verfahren soll nicht Teil dieser Arbeit sein.

Bei der Implementierung dieser Methode treten keine neuen Probleme mit dem Ownership und Borrowing von Rust auf. Es ist jederzeit nachvollziehbar, an welchen Stellen Werte geklont werden müssen. So müssen z.B. bei der Übergabe der Parameter bei einem rekursiven Funktionsaufruf diese geklont werden, da ansonsten entsprechend das Ownership abgegeben wird. Die bisherigen Implementierungen von `traits` zur Prüfung auf Gleichheit mit Ganzzahlen von `Ast` und `NumberType` sind auch hier ausreichend.

Als Ergänzung wurde die ∞-Konstante hinzugefügt, die nun auch während des Parsens erkannt wird. Darüber hinaus wurde die `shorten`-Methode erweitert, um `Add`- und `Mul`-Vektoren entsprechend zu behandeln, sofern die ∞-Konstante vorhanden ist. So wird z.B. $2 * \infty$ zu ∞ vereinfacht.

7.3 Terme leserlich darstellen

Bisher wurde zur Ausgabe des Terms im `println!`-Makro der Platzhalter "{:?}" verwendet. Dieser erlaubt es `structs` und `enums` ohne weitere Formatierung auszugeben. Im vorliegenden Fall wird z.B. für den Term $-1 + y + x * 2 + x^2$ die Ausgabe

> Add([Mul([Num(Int(2)), Symbol("x")]), Pow(Symbol("x"), Num(Int(2))), Symbol("y"), Num(Int(-1))])

erzeugt. Da dies noch unleserlich ist und die übliche Notation die Potenz an den Anfang stellen würde, wird der Trait `Display` implementiert.

Sobald dieser implementiert ist, kann die Ausgabe auch mit `println!("{}", ast)` erfolgen. Für das Trait muss die Methode `fmt` überladen werden, die dann aufgerufen wird. Hier können die Vektoren von `Add` und `Mul` entsprechend sortiert werden, sodass eine übliche Ausgabe erfolgt. Die Implementierung in dieser Arbeit gibt für den oben genannten Term $x^2 + 2 * x + y - 1$ aus.

8 Vergleich zu SymPy

In diesem Kapitel sollen die Korrektheit und Performance überprüft werden. Hierzu werden einige Beispielrechnungen festgelegt und ausgeführt. Ein Vergleich zu SymPy lässt eine Einordnung zu anderen Systemen zu.

8.1 Korrektheit und Unterschiede gegenüber SymPy

Um beurteilen zu können, ob das entwickeltes System richtig rechnet, soll mit verschiedenen Termen geprüft werden, ob SymPy dieselben Ergebnisse berechnet. Unterschiede zwischen Ergebnissen werden näher betrachtet.

Tabelle 2: Numerische Berechnungen

Term	**Ergebnis CAS**	**Ergebnis SymPy**
$1+2+3+4-5$	5	5
$1*2*3*4$	24	24
$1*2*3*4*0$	0	0
$1*2*3*4*(-1)$	-24	-24
$2+3*4+5$	19	19
$(2+3)*(4+5)$	45	45
2^{3^4}	2417851639229258349412352	2417851639229258349412352
$0.1+0.2$	0.30000000000000004	0.300000000000000
$3^{-1}+3^{-1}$	$\frac{2}{3}$	$\frac{2}{3}$
$3^{-1}*3^{-1}$	$\frac{1}{9}$	$\frac{1}{9}$
$\sqrt{2}$	$2^{\frac{1}{2}}$	$\sqrt{2}$
$\sqrt{2}+\sqrt{2}$	$2*2^{\frac{1}{2}}$	$2*\sqrt{2}$
$\sqrt{2}*\sqrt{2}$	2	2

Bei dieser Gegenüberstellung fallen zwei Unterschiede auf. Zum einen tritt das anfangs beschriebene Problem der Ungenauigkeit auf. Eine Lösung hierfür ist die Benutzung von PrecisionNum.

Codebeispiel 35: PrimNum vs. PrecisionNum

```
let a1 = Ast::Num(PrimNum::Float(0.1));
let a2 = Ast::Num(PrimNum::Float(0.2));
println!("{}", a1 + a2);
// 0.30000000000000004

let a1 = Ast::Num(PrecisionNum::Float(
        BigDecimal::from_str("0.1").unwrap()));
let a2 = Ast::Num(PrecisionNum::Float(
        BigDecimal::from_str("0.2").unwrap()));
println!("{}", a1 + a2);
// 0.3
```

Der zweite Unterschied ist die Ausgabe von $\sqrt{2}$. In implementierten System wird diese als Potenz ausgegeben, in SymPy als Funktion. Diese Darstellung ist jedoch nur kosmetisch. Wird der Term in SymPy mit `sympy.srepr(expr)` ausgegeben, ist die Struktur des Terms zu sehen. SymPy wandelt die *sqrt*-Funktion ebenfalls in eine Potenz um und gibt diese in diesem speziellen Fall formatiert aus.

Tabelle 3: Terme mit Variablen zusammenfassen

Term	**Ergebnis CAS**	**Ergebnis SymPy**
$x + x + 2 * x$	$4 * x$	$4 * x$
$2 * x - x - x + y$	y	y
$x + x + a * x$	$a * x + 2 * x$	$a * x + 2 * x$
$a * x - x - x + y$	$a * x - 2 * x + y$	$a * x - 2 * x + y$
$x * 0$	0	0
$x * x$	x^2	x^2
$x^2 * x$	x^3	x^3
$x^a * x$	x^{a+1}	$x * x^a$
x/x	1	1
x^2/x	x	x
x^a/x	x^{a-1}	x^a/x
$a * x + b * x$	$a * x + b * x$	$a * x + b * x$
$a^x * b^x$	$a^x * b^x$	$a^x * b^x$
$x^a * x^b$	x^{a+b}	$x^a * x^b$

Bei der Zusammenfassung von Variablen tritt ein Unterschied bei den Termen $x^a * x$, x^a/x und $x^a * x^b$ auf. In diesem System werden, durch die implementierte Logik, vermehrt Potenzen zusammengefasst, was in SymPy standardmäßig nicht der Fall ist.

Tabelle 4: Terme mit Variablen substituieren

Term	**Ergebnis CAS**	**Ergebnis SymPy**
$2 + x$, $x = 3$	5	5
$2 + x$, $x = -3$	-1	-1
$2 - x$, $x = -3$	5	5
$2 + x$, $x = 0.5$	2.5	2.50000000000000
$2^x + 2^x$, $x = 2$	8	8
$2^x + 2^x$, $x = -1$	1	1
$\sqrt{x}$, $x = 3$	$3^{\frac{1}{2}}$	$sqrt(3)$
$\sqrt{x}$, $x = 4$	2	2
$\sqrt{x}$, $x = 16$	4	4
$\sqrt{x}$, $x = 17$	$17^{\frac{1}{2}}$	$sqrt(17)$

Die Substituierung wird ebenfalls identisch umgesetzt. Einzig die Ausgabe bei Termen mit Wurzeln ist wie bereits erwähnt unterschiedlich.

Tabelle 5: Nutzung der Methode “expand”

Term	**Ergebnis CAS**	**Ergebnis SymPy**
$(x+1)*(x-2)-(x-1)*x$	-2	-2
$(a+b)*(c+d)*e$	$a*c*e+a*d*e$ $+b*c*e+b*d*e$	$a*c*e+a*d*e$ $+b*c*e+b*d*e$
$(a+b)^3$	$a^3+b^3+3*a^2*b$ $+3*b^2*a$	$a^3+3*a^2*b+3*a*b^2$ $+b^3$
x^{a+b}	x^a*x^b	x^a*x^b
x^{a*b}	x^{a*b}	x^{a*b}
$(x+y)^a$	$(x+y)^a$	$(x+y)^a$
$(x*y)^a$	x^a*y^a	x^a*y^a
$log(x^2*y)$	$2*log(x)+log(y)$	$2*log(x)+log(y)$
$log(x^a*y)$	$a*log(x)+log(y)$	$a*log(x)+log(y)$

Terme, die mit “expand” in beiden Systemen verarbeitet werden, liefern jeweils gleiche Ergebnisse. Dies ist natürlich nur der Fall, für die in diesem System implementierten Funktionen. Andere mathematische Funktionen, die von SymPy korrekt verarbeitet werden, müssten erst nachimplementiert werden. So werden in SymPy ebenfalls trigonomische Funktionen ausgewertet. So wird aus $sin(2*x)$ der Term $2*sin(x)*cos(x)$ [74].

Tabelle 6: Nutzung der Methode “simplify”

Term	**Ergebnis CAS**	**Ergebnis SymPy**
$(x^2+x)/x$	$x+1$	$x+1$
$(x^2+x)*(y^2+y)/x/y$	$(y+1)*(x+1)$	$(x+1)*(y+1)$
$(x^2+x)*(y^2+y)/x/a$	$a^-1*(x+1)*(y^2+y)$	$y*(x+1)*(y+1)/a$
$(x^3+x^2+x)*(x^2+x)/x$	$(x^2+x)*(x^2+x+1)$	$x*(x+1)*(x^2+x+1)$

Auch bei der Nutzung von “simplify” werden gleichwertige Ergebnisse erzeugt. SymPy klammert zusätzlich aber noch gemeinsame Teiler aus, was das vorliegende System nicht beherrscht. Dies ist vor allem am dritten Beispiel zu erkennen. Beide Terme sind aber in diesem Fall korrekt. Dieser Fall wurde bereits bei der Implementierung von “simplify” erwähnt.

8.2 Performancevergleich

Bei diesem Vergleich soll die Laufzeit, der Speicherverbrauch und die CPU-Auslastung in verschiedenen Szenarien verglichen werden. Die verwendete Hardware besteht aus einem “AMD Ryzen 5 3600X 6-Core Processor” und 16 GB Arbeitsspeicher mit 3000 MHz (DDR 4). Als Messtool wird die Windows-Version des “Benchmarks Game” verwendet [25] [26].

Python liegt hierbei aus den offiziellen Quellen in Version 3.9.7 vor [62]. Der Rust-Compiler besitzt die Version “rustc 1.60.0” und wurde mithilfe von “rustup” installiert [64]. Die verwendeten ausführbaren Programme wurden als Release-Version kompiliert und so geschrieben, dass der Compiler die Berechnungen bei der Optimierung des Codes nicht entfernt.

Alle Szenarien werden zehn mal ausgeführt und dargestellt. Die Zeiten sind in Sekunden angegeben, der Speicherverbrauch in Kilobyte und die CPU-Auslastung in Prozent.

1. Szenario Hier werden alle Berechnungen des vorherigen Kapitels durchgeführt. Alle Terme werden dabei sowohl in SymPy als auch in Rust aus einer Zeichenkette geparst und entsprechend ausgeführt.

Tabelle 7: Performance Tests

Lauf	Python Zeit	Speicher	CPU	Rust Zeit	Speicher	CPU
1	0.625	40560	6	0.000	756	0
2	0.688	40488	6	0.000	764	0
3	0.641	40548	6	0.016	768	0
4	0.641	40692	6	0.000	752	0
5	0.609	40576	6	0.000	756	0
6	0.641	40324	6	0.000	756	0
7	0.609	40636	5	0.000	752	0
8	0.641	40648	7	0.000	752	0
9	0.594	40808	6	0.016	772	0
10	0.641	40712	6	0.000	756	0

2. Szenario Der Term $a + b + c$ wird einmalig erstellt. Hiernach werden drei geschachtelte for-Schleifen durchlaufen, jeweils von 0 bis 49. Jede Variable wird hierbei mit einem Schleifenzähler substituiert. Die erste Substitution ist somit $0 + 0 + 0$, die zweite $0 + 0 + 1$ und die letzte $49 + 49 + 49$.

Tabelle 8: Performance Addition

Lauf	Python Zeit	Speicher	CPU	Rust Zeit	Speicher	CPU
1	9.781	37796	8	0.453	740	7
2	9.656	37612	8	0.469	732	8
3	9.859	37768	8	0.438	732	8
4	9.688	37760	8	0.453	752	8
5	9.734	37960	8	0.469	756	8
6	9.766	37696	8	0.453	744	8
7	9.875	37740	8	0.469	712	8
8	9.594	37920	8	0.438	736	8
9	9.609	37912	8	0.453	740	8
10	9.688	37964	8	0.469	736	8

3. Szenario Der Term $a * b * c$ wird einmalig erstellt. Hiernach werden drei geschachtelte for-Schleifen durchlaufen, jeweils von 0 bis 49. Jede Variable wird hierbei mit einem Schleifenzähler substituiert. Die erste Substitution ist somit $0 * 0 * 0$, die zweite $0 * 0 * 1$ und die letzte $49 * 49 * 49$.

Tabelle 9: Performance Multiplikation

Lauf	Python Zeit	Speicher	CPU	Rust Zeit	Speicher	CPU
1	12.594	37996	8	0.344	792	7
2	12.812	37460	8	0.344	744	8
3	12.906	37852	8	0.344	740	8
4	12.797	37596	8	0.344	740	8
5	12.719	37508	8	0.359	744	8
6	12.812	37856	8	0.344	752	8
7	12.922	37536	8	0.344	744	8
8	12.547	37744	8	0.344	744	8
9	12.672	37424	8	0.328	752	8
10	12.922	37576	8	0.344	744	7

4. Szenario In diesem Szenario liegt der Fokus auf den Speicherverbrauch. Es wird wieder der Term $a + b + c$ einmalig erstellt. Hiernach werden drei geschachtelte for-Schleifen durchlaufen, jeweils von 0 bis 49. Einer am Anfang erstellten Liste bzw. Vektor wird sowohl der originale Term als auch die Substituierung wie im 2. Szenario hinzugefügt.

Tabelle 10: Speicherverbrauch

Lauf	Python Zeit	Speicher	CPU	Rust Zeit	Speicher	CPU
1	9.703	42608	8	0.578	43792	7
2	9.891	41680	8	0.484	43748	7
3	9.797	42496	8	0.500	43784	7
4	9.578	41668	8	0.516	43788	7
5	9.750	42444	8	0.500	43752	8
6	9.719	42280	8	0.500	43796	8
7	9.781	42448	8	0.516	43792	7
8	9.734	41792	8	0.500	43744	7
9	9.750	42360	8	0.516	43736	7
10	9.656	42552	8	0.516	43792	7

Beobachtungen Bei einem Vergleich der Szenarien 1 bis 3 ist wie zu erwarten, das kompilierte Programm sowohl schneller als auch speichereffizienter. Der geringe Speicherverbrauch in Rust lässt sich einerseits auf das automatische Deallokieren von Variablen zurückführen, wenn diese den Gültigkeitsbereich verlassen. Andererseits gibt es keine Laufzeitumgebung wie in Python. Bei den SymPy-Tests dagegen ist der große Sprung zum Speicherplatzverbrauch dagegen auf die SymPy-Bibliothek zurückzuführen.

Wird in Python ein einfaches “Hello World!” ausgegeben, beträgt der Speicherverbrauch ca. 5000 Kilobyte. Wird im selben Python-Skript zusätzlich SymPy importiert, beträgt der Speicherverbrauch bereits ca. 35000 Kilobyte. Da zudem unklar ist, wann genau der Garbage Collector von Python läuft, kann es auch vorkommen, dass noch ungenutzte Variablen den Speicher belegen.

Im ausführbaren Programm von Rust, sind dagegen nur Codeabschnitte, die tatsächlich benögit werden. Nicht benötigte Funktionen aus Abhängigkeiten und dem eigenen Code sind nicht vorzufinden [85]. Dies kann man sich auch anschaulich mit “cargo-bloat” anzeigen lassen [63].

Bei Szenario 4 zeigt die Laufzeit wieder ein ähnliches Verhältnis. Rust ist wieder schneller als SymPy. Auf den ersten Blick skaliert aber SymPy wesentlich besser in Bezug auf den Speicherplatzverbrauch. Dies ist aber stark implementierungsabhängig. Da wie anfänglich beschrieben, immer der originale Term und das Ergebnis der Liste hinzugefügt wird, muss aufgrund des Ownerships in Rust der originale Vektor immer kopiert werden. da dieser nur einmalig erstellt wird. Bei dem Hinzufügen zur Liste wird folglich das Ownership verschoben.

Wird in der Python-Version anstelle des originalen Terms die Adresse der Variablen der Liste hinzugefügt und lässt sich diese ausgeben, fällt auf, dass immer nur eine Referenz hinzugefügt wird, da alle Adressen identisch sind. Lediglich die Ergebnisse unterscheiden sich. Wird der originale Term nicht der Liste hinzugefügt, beträgt der Speicherplatzverbrauch von Rust nur noch ca. 13000 Kilobyte, die Python-Version belegt immer noch ca. 40000 Kilobyte.

Anstelle einer kopierten Version des Terms könnte auch eine Referenz der Liste hinzugefügt werden. Jedoch muss sichergestellt werden, dass der Term eine größere Lifetime als der Vektor besitzt.

Aus den Ergebnissen kann geschlossen werden, dass Rust im Vergleich zu SymPy eine wesentlich bessere Performance bietet. Dies betrifft zum einen die Laufzeit von Berechnungen, als auch den Speicherplatzverbrauch. Letzteres hängt jedoch von den implementierten und verwendeten Funktionen des Systems ab.

9 Fazit

9.1 Erweiterungs- und Verbesserungsideen des Systems

Im Rahmen dieser Arbeit wurde aufgrund des Umfangs nicht für alle Implementierungen die bestmögliche Lösung vorgenommen. Auch sind einige Verbesserungen möglich, die für einen angenehmeren Umgang mit dieser Bibliothek sorgen würden. Diese Punkte sollen der Vollständigkeit halber mit aufgeführt werden.

- Implementierung einer iterativen Lösung um Elemente des `Ast` zu verarbeiten.
- Der Parser könnte erweitert werden, sodass auch $3x$ anstelle von $3 * x$ erkannt wird. Auch die Möglichkeit dynamisch Operatoren hinzuzufügen wäre interessant, sodass z.B. für die Fakultät einfach $x!$ angegeben werden kann, anstelle eines Funktionsaufrufs.
- Bei der Zusammenfassung der Terme ist zu überlegen, den Nummerntyp nicht zusammenzufassen. Rationale Zahlen und Fließkommazahlen könnten nebeneinander existieren. In der momentanen Implementierung wird diese Addition zu einer Fließkommazahl vereinfacht. Der Vorteil dabei ist, dass sicher nur ein Zahlenwert vorhanden ist. Allerdings wird der Nachteil der Ungenauigkeit in Kauf genommen.
- Bei der Zusammenfassung von Termen in der HashMap kann es zu Konflikten kommen. Eine andere Idee wäre, als Schlüssel den Term als String zu verwenden. Hierzu müsste jedoch 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.
- Im Moment wird $cos(x)^2 + sin(x)^2 = 1$ nur in einfachen Fällen zusammengefasst. Der Term $(z * sin(x)^2) + (z * cos(x)^2) + (a * sin(x)^2) + (a * cos(x)^2)$ könnte entsprechend auch zusammengefasst werden.
- Erweiterung des heuristischen Verfahrens der “limit”-Funktion und Implementierung des Gruntz-Algorithmus [27].

Bei der Substituierung von Symbolen wären ebenfalls Erweitungen vorstellbar.

- Übergabe eines Vektors von Werten zur Berechnung. Der Rückgabewert ist ebenfalls ein Vektor mit den entsprechenden Substituierungen. Diese könnten sogar nebenläufig umgesetzt werden.
- Mehrere Variablen mit einem Funktionsaufruf ersetzen.
- Substitution von ganzen Termen erlauben, nicht nur einzelnen Variablen.
- Mischformen der Verbesserungen.

9.2 Ausblick

Am Anfang dieser Arbeit waren die Konzepte Ownership, Borrowing und Lifetimes noch schwer greifbar. Diese wurden als Hürde wahrgenommen, da hierdurch einige Implementierungen explizit angegeben werden müssen, wie z.B. das Klonen von Übergabeparametern oder auch wie teilweise über Elemente iteriert werden muss. Im Laufe der Entwicklung des CAS wurden diese Konstrukte aber immer augenscheinlicher und stellen schlussendlich keine Probleme mehr dar. Darüber hinaus kann immer leicht nachvollzogen werden, an welchen Stellen, welche Werte verwendet werden. Zusätzlich stellt Rust sicher, dass die Werte nicht

versehentlich an anderer Stelle überschrieben worden sind. Die Fehleranfälligkeit sinkt dadurch, wie in Kapitel 3.2 gezeigt.

Dieses strenge System in Rust hat sich von einem Nach- zu einem großen Vorteil entwickelt. Verbunden mit einer steilen Lernkurve von Rust sind dies die einzigen Nachteile, die während der Arbeit mit Rust aufgefallen sind.

Ein weiterer Vorteil von Rust ist die bessere Performance im Gegensatz zu SymPy. Dieser Vorteil dürfte im Vergleich zu anderen Systemen, die in C oder C++, wie z.B. Xcas [88] geschrieben sind, geringer ausfallen. Das hängt im konkreten Fall vom verwendeten Compiler ab, der die ausführbaren Programme erzeugt. Da Rust, wie bereits angesprochen LLVM verwendet, würde ein C/C++ Programm, kompiliert mit Clang wahrscheinlich eine ähnliche Performance aufweisen.

Ebenfalls ein Vorteil ist das breite Ökosystem, welches mit “cargo” bereitsteht. Das Programm kann einfach mit verschiedenen Profilen kompiliert werden, z.B. als Debug- oder Releaseversion. Abhängigkeiten können installiert und Tests und Bechnmarks ausgeführt werden.

Auch wenn, wie Eingangs erwähnt, Rust derzeit keine große Relevanz im Bereich des wisschenschaftlichen Arbeitens besitzt, so sprechen die genannten Punkte dafür, Rust auch dort zu verwenden. Wie ebenfalls in der Einleitung erwähnt, wird die Sprache in anderen Projekten bereits erfolgreich eingesetzt.

Gerne veröffentliche ich nach Absprache mit Frau Prof. Dr. Oden die entstandene Arbeit und stelle diese der Rust-Community zur Verfügung. Eine zukünftige Weiterentwicklung soll nicht ausgeschlossen werden.

Literatur

[1] Crate bigdecimal: *Crate bigdecimal*. https://crates.io/crates/bigdecimal, 2022. [abgerufen am 18.07.2022].

[2] Crate bigdecimal: *Crate bigdecimal Docs*. https://docs.rs/bigdecimal/latest/bigdecimal/, 2022. [abgerufen am 18.07.2022].

[3] Jim Blandy und Jason Orendorff: *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, Inc, 1. Auflage, 2018, ISBN 978-1-491-92728-1.

[4] Rust Blog: *Rust Lang Roadmap for 2024*. https://blog.rust-lang.org/inside-rust/2022/04/04/lang-roadmap-2024.html, 2022. [abgerufen am 18.07.2022].

[5] Rust Blog: *Theme: Flatten the (learning) curve*. https://blog.rust-lang.org/inside-rust/2022/04/04/lang-roadmap-2024.html#theme-flatten-the-learning-curve, 2022. [abgerufen am 18.07.2022].

[6] Rust Source Code: *Arc struct*. https://doc.rust-lang.org/src/alloc/sync.rs.html#87-94. [abgerufen am 18.07.2022].

[7] Rust Source Code: *String struct*. https://doc.rust-lang.org/src/alloc/string.rs.html#294-296. [abgerufen am 18.07.2022].

[8] codeplea: *tinyexpr Readme*. https://github.com/codeplea/tinyexpr/blob/master/README.md#te_compile-te_eval-te_free, 2022. [abgerufen am 18.07.2022].

[9] codeplea: *tinyexpr Source*. https://github.com/codeplea/tinyexpr, 2022. [abgerufen am 18.07.2022].

[10] Fachgruppe Computeralgebra: *Anwendungen der Computeralgebra*. https://fachgruppe-computeralgebra.de/computeralgebra/. [abgerufen am 18.07.2022].

[11] Fachgruppe Computeralgebra: *Was ist Computeralgebra?* https://fachgruppe-computeralgebra.de/computeralgebra/. [abgerufen am 18.07.2022].

[12] ndarray developers: *ndarray*. https://docs.rs/ndarray/latest/ndarray/, 2022. [abgerufen am 18.07.2022].

[13] dimforge: *nalgebra*. https://github.com/dimforge/nalgebra, 2022. [abgerufen am 18.07.2022].

[14] Python docs: *Numeric Types - int, float, complex*. https://docs.python.org/3.8/library/stdtypes.html#numeric-types-int-float-complex. [abgerufen am 18.07.2022].

[15] SymPy Docs: *Understanding Expresion Trees*. https://docs.sympy.org/latest/tutorial/manipulation.html#understanding-expression-trees, 2022. [abgerufen am 18.07.2022].

[16] Rust Editions: *Rust 2021*. https://doc.rust-lang.org/edition-guide/rust-2021/index.html, 2022. [abgerufen am 18.07.2022].

[17] Crate evalexpr: *Crate evalexpr*. https://crates.io/crates/evalexpr, 2022. [abgerufen am 18.07.2022].

[18] Rust By Example: *Operator Overloading*. https://doc.rust-lang.org/rust-by-example/trait/ops.html, 2022. [abgerufen am 18.07.2022].

[19] Rust By Example: *Ownership and moves*. https://doc.rust-lang.org/rust-by-example/scope/move.html, 2022. [abgerufen am 18.07.2022].

[20] fasterthanlime: *Peeking inside a Rust enum*. https://fasterthanli.me/articles/peeking-inside-a-rust-enum, 2020. [abgerufen am 18.07.2022].

[21] Felipe: *Using Rust for Scientific Numerical applications: Learning from Past Experiences*. https://blog.esciencecenter.nl/798665d9f9f0, 2021. [abgerufen am 18.07.2022].

[22] Sachan Ganesh: *Graph & Tree Traversals in Rust*. https://sachanganesh.com/programming/graph-tree-traversals-in-rust/, 2022. [abgerufen am 18.07.2022].

[23] Joachim von zur Gathen und Jürgen Gerhard: *Modern Computer Algebra* -, 2013, ISBN 978-1-107-03903-2.

[24] Keith O. Geddes, Stephen R. Czapor und George Labahn: *Algorithms for Computer Algebra* -, 2007, ISBN 978-0-585-33247-5.

[25] Isaac Gouy: *Bechmarks Game Website*. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html, 2022. [abgerufen am 18.07.2022].

[26] Isaac Gouy: *Bechmarks Game Windows Src*. https://salsa.debian.org/benchmarksgame-team/benchmarksgame/-/blob/master/bencher/bin/planBwin32.py, 2022. [abgerufen am 18.07.2022].

[27] Dominik Gruntz: *On Computing Limits in a Symbolic Manipulation System*. Dissertation, Swiss Federal Institute of Technology Zurich, 1996.

[28] Ralf Hartmut Güting: *Übersetzerbau*. FernUniversität Hagen Kurs 01810, Version Wintersemester 2017/2018.

[29] Rust Edition Guide: *Rust Edition Guide*. https://doc.rust-lang.org/edition-guide/introduction.html. [abgerufen am 18.07.2022].

[30] Rustc dev guide: *Code generation*. https://rustc-dev-guide.rust-lang.org/backend/codegen.html#code-generation, 2022. [abgerufen am 18.07.2022].

[31] Rustc dev guide: *What the compiler does to your code*. `https://rustc-dev-guide.rust-lang.org/overview.html#what-the-compiler-does-to-your-code`, 2022. [abgerufen am 18.07.2022].

[32] Bernd Haßfurther: *Die Programmiersprache Rust*, 2021.

[33] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019 (Revision of IEEE 754-2008), Seiten 1–84, 2019.

[34] Prof. Dr. Jörg Keller: *Parallel Programming*. FernUniversität Hagen Kurs 01727, Version Wintersemester 2019/2020.

[35] Steve Klabnik und Carol Nichols: *The Borrow Checker*, Kapitel Validating References with Lifetimes. No Starch Press, 2019, ISBN 9781718500440.

[36] Steve Klabnik und Carol Nichols: *Enabling Recursive Types with Boxes*, Kapitel Using Box<T> to Point to Data on the Heap. No Starch Press, 2019, ISBN 9781718500440.

[37] Steve Klabnik und Carol Nichols: *Ownership Rules*, Kapitel What Is Ownership? No Starch Press, 2019, ISBN 9781718500440.

[38] Steve Klabnik und Carol Nichols: *Ownership Rules*, Kapitel What Is Ownership? No Starch Press, 2019, ISBN 9781718500440.

[39] Steve Klabnik und Carol Nichols: *Performance of Code Using Generics*, Kapitel Generic Data Types. No Starch Press, 2019, ISBN 9781718500440.

[40] Steve Klabnik und Carol Nichols: *References and Borrowing*, Kapitel References and Borrowing. No Starch Press, 2019, ISBN 9781718500440.

[41] Steve Klabnik und Carol Nichols: *The Rules of References*, Kapitel References and Borrowing. No Starch Press, 2019, ISBN 9781718500440.

[42] Steve Klabnik und Carol Nichols: *The Stack and the Heap*, Kapitel What Is Ownership? No Starch Press, 2019, ISBN 9781718500440.

[43] Steve Klabnik und Carol Nichols: *Stack-Only Data: Copy*, Kapitel What Is Ownership? No Starch Press, 2019, ISBN 9781718500440.

[44] Steve Klabnik und Carol Nichols: *Unsafe Rust*, Kapitel Unsafe Rust. No Starch Press, 2019, ISBN 9781718500440.

[45] Steve Klabnik und Carol Nichols: *Using Box<T> to Point to Data on the Heap*, Kapitel Using Box<T> to Point to Data on the Heap. No Starch Press, 2019, ISBN 9781718500440.

[46] Steve Klabnik und Carol Nichols: *Using Trait Objects That Allow for Values of Different Types*, Kapitel Using Trait Objects That Allow for Values of Different Types. No Starch Press, 2019, ISBN 9781718500440.

[47] Steve Klabnik und Carol Nichols: *Validating References with Lifetimes*, Kapitel Validating References with Lifetimes. No Starch Press, 2019, ISBN 9781718500440.

[48] Steve Klabnik und Carol Nichols: *Variables and Mutability*, Kapitel Variables and Mutability. No Starch Press, 2019, ISBN 9781718500440.

[49] Edmund A. Lamagna: *Computer Algebra - Concepts and Techniques*, 2019, ISBN 978-1-351-60583-0.

[50] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman und Anthony Scopatz: *SymPy: symbolic computing in Python.* PeerJ Computer Science, 3:e103, Januar 2017, ISSN 2376-5992. `https://doi.org/10.7717/peerj-cs.103`.

[51] Crate meval: *Crate meval.* `https://crates.io/crates/meval`, 2022. [abgerufen am 18.07.2022].

[52] Crate mexprp: *Crate mexprp.* `https://crates.io/crates/mexprp`, 2022. [abgerufen am 18.07.2022].

[53] Microsoft: *windows-rs.* `https://github.com/microsoft/windows-rs`. [abgerufen am 18.07.2022].

[54] Robert Neumann: *Zum Einfluss von Computeralgebrasystemen auf mathematische Grundfertigkeiten - Eine empirische Bestandsaufnahme.* Springer Spektrum, 1. Auflage, 2018, ISBN 978-3-658-18949-5.

[55] Crate num: *BigInt Sign.* `https://github.com/rust-num/num-bigint/blob/master/src/bigint.rs#L41`, 2022. [abgerufen am 18.07.2022].

[56] Crate num: *Crate num.* `https://crates.io/crates/num`, 2022. [abgerufen am 18.07.2022].

[57] Miguel Ojeda: *Rust support.* `https://lore.kernel.org/lkml/20220507052451.12890-1-ojeda@kernel.org/`, 2022. [abgerufen am 18.07.2022].

[58] Oracle: *Type Erasure.* `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`. [abgerufen am 18.07.2022].

[59] lumol org: *lumol.* `https://github.com/lumol-org/lumol`, 2022. [abgerufen am 18.07.2022].

[60] Jeffrey M. Perkel: *Why scientists are turning to Rust.* `https://www.nature.com/articles/d41586-020-03382-2`, 2020. [abgerufen am 18.07.2022].

[61] PoisonAlien: *hapcounter.* `https://github.com/CompEpigen/hapcounter`, 2022. [abgerufen am 18.07.2022].

[62] Python: *Python Download.* `https://www.python.org/downloads/`, 2022. [abgerufen am 18.07.2022].

[63] RazrFalcon: *cargo-bloat.* `https://github.com/RazrFalcon/cargo-bloat`, 2022. [abgerufen am 18.07.2022].

[64] Rust: *Rust Download*. https://www.rust-lang.org/tools/install, 2022. [abgerufen am 18.07.2022].

[65] Rahul Sharma und Vesa Kaihlavirta: *Mastering RUST*. Packt Publishing Ltd., second edition Auflage, 2019, ISBN 978-1-78934-657-2.

[66] ChemPy Src: *ChemPy README*. https://github.com/bjodah/chempy/blob/master/README.rst, 2022. [abgerufen am 18.07.2022].

[67] SymPy src: *class Float*. https://github.com/sympy/sympy/blob/master/sympy/core/numbers.py#L856, 2022. [abgerufen am 18.07.2022].

[68] SymPy src: *flatten Add*. https://github.com/sympy/sympy/blob/master/sympy/core/add.py#L185, 2022. [abgerufen am 18.07.2022].

[69] SymPy src: *flatten Mul*. https://github.com/sympy/sympy/blob/master/sympy/core/mul.py#L198, 2022. [abgerufen am 18.07.2022].

[70] SymPy src: *Newton's method*. https://github.com/sympy/sympy/blob/master/sympy/core/power.py#L99, 2022. [abgerufen am 18.07.2022].

[71] SymPy src: *Notes for limit function*. https://github.com/sympy/sympy/blob/master/sympy/series/limits.py#L51-L56, 2022. [abgerufen am 18.07.2022].

[72] SymPy src: *simplify*. https://github.com/sympy/sympy/blob/master/sympy/simplify/simplify.py#L420, 2022. [abgerufen am 18.07.2022].

[73] SymPy Src: *SymPy first commit*. https://github.com/sympy/sympy/commit/99b21ff58ad2e2ba83172512d7f513a7c37e50c3, 2022. [abgerufen am 18.07.2022].

[74] SymPy src: *Trigonomische Funktionen mit expand*. https://github.com/sympy/sympy/blob/master/sympy/core/function.py#L2640, 2022. [abgerufen am 18.07.2022].

[75] Rust std: *Rust Eq Trait*. https://doc.rust-lang.org/std/cmp/trait.Eq.html, 2022. [abgerufen am 18.07.2022].

[76] Rust std: *Rust i64 impl Copy*. https://doc.rust-lang.org/std/primitive.i64.html#impl-Copy, 2022. [abgerufen am 18.07.2022].

[77] Rust std: *Struct Cell*. https://doc.rust-lang.org/stable/std/cell/struct.Cell.html, 2022. [abgerufen am 18.07.2022].

[78] Rust std: *Struct Rc*. https://doc.rust-lang.org/stable/std/rc/struct.Rc.html, 2022. [abgerufen am 18.07.2022].

[79] Rust std: *Struct RefCell*. https://doc.rust-lang.org/stable/std/cell/struct.RefCell.html, 2022. [abgerufen am 18.07.2022].

[80] Jeff Vander Stoep und Stephen Hines: *Rust in the Android platform*. https://security.googleblog.com/2021/04/rust-in-android-platform.html, 2022. [abgerufen am 18.07.2022].

[81] SymPy: *SymPy About*. https://www.sympy.org/en/index.html. [abgerufen am 18.07.2022].

[82] Kiat Shi Tan, Willi Hans Steeb und Yorick Hardy: *SymbolicC++: An Introduction to Computer Algebra using Object-Oriented Programming*, 2000, ISBN 978-1-852-33260-0.

[83] Rust Team: *Rust Promises*. https://www.rust-lang.org/. [abgerufen am 18.07.2022].

[84] SymPy Development Team: *expand*. https://docs.sympy.org/latest/tutorial/simplification.html#expand, 2021. [abgerufen am 18.07.2022].

[85] Rust users: *Rust Dead code elimination for libraries*. https://users.rust-lang.org/t/dead-code-elimination-for-libraries/28217, 2022. [abgerufen am 18.07.2022].

[86] Werne Vogt: *Zur Numerik nichtlinearer Gleichungssysteme (Teil 1)*. Preprint https://www.db-thueringen.de/servlets/MCRFileNodeServlet/dbt_derivate_00008992/IfM_Preprint_M_01_12.pdf#page=30&zoom=auto,-13,488, Oktober 2001.

[87] sagemath Website: *sagemath*. https://www.sagemath.org/, 2022. [abgerufen am 18.07.2022].

[88] Xcas: *Xcas Website*. http://www-fourier.ujf-grenoble.fr/~parisse/giac.html, 2022. [abgerufen am 18.07.2022].

[89] Daniel Zwillinger: *List of commont limits*, Kapitel Differential Calculus. Crc Press, 1996.