Bachelorarbeit

Ein Computeralgebrasystem in Rust

Verfasser	Bernd Haßfurther <nerglom@posteo.de>
Matrikel-Nr.	4372280
Betreuerin	Prof. Dr. Lena Oden
Datum	30. August 2022

Inhaltsverzeichnis

Was ist ein CAS?

- Mathematische Ausdrücke mit Variablen darstellen

Was ist ein CAS?

- Mathematische Ausdrücke mit Variablen darstellen
- Resistent gegen Ungenauigkeiten

Was ist ein CAS?

- Mathematische Ausdrücke mit Variablen darstellen
- Resistent gegen Ungenauigkeiten
- Einsatz in verschiedenen Gebieten

Was ist ein CAS?

- Mathematische Ausdrücke mit Variablen darstellen
- Resistent gegen Ungenauigkeiten
- Einsatz in verschiedenen Gebieten
- SymPy als konkrete Implementierung

(vgl. [2] [8, S. 1])

Grundregeln und Annahmen

- Zahlenräume $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$

Grundregeln und Annahmen

- Zahlenräume $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$
- Jede Subtraktion ist eine Addition

Grundregeln und Annahmen

- Zahlenräume $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$
- Jede Subtraktion ist eine Addition
- Jede Division ist entweder eine rationale Zahl oder eine Multiplikation

Grundregeln und Annahmen

- Zahlenräume $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$
- Jede Subtraktion ist eine Addition
- Jede Division ist entweder eine rationale Zahl oder eine Multiplikation
- Wurzeln können als Potenzen dargestellt werden

(vgl. [3, S. 23 ff.] [11, S. 2])

Funktionsumfang der Implementierung

- Addition von Zahlen und Symbolen

Funktionsumfang der Implementierung

- Addition von Zahlen und Symbolen
- Multiplikation von Zahlen und Symbolen

Funktionsumfang der Implementierung

- Addition von Zahlen und Symbolen
- Multiplikation von Zahlen und Symbolen
- Potenzregeln in Hinblick auf Genauigkeit auswerten

Funktionsumfang der Implementierung

- Addition von Zahlen und Symbolen
- Multiplikation von Zahlen und Symbolen
- Potenzregeln in Hinblick auf Genauigkeit auswerten
- Auswertung von mathematischen Funktionen und Konstanten

Ziele von Rust

- Performance

Ziele von Rust

- Performance
- Verlässlichkeit

Ziele von Rust

- Performance
- Verlässlichkeit
- Produktivität

(vgl. [12] [10, S. 196 ff.] [9])

Hinweise zur Syntax

- Expression und Statements

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

Hinweise zur Syntax

- Expression und Statements

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

- `struct` und `trait` als Klassen und Interfaces

```
struct MyStruct {  my_field: i32, }
impl MyStruct { fn do_smth(&self) {} }
```

Hinweise zur Syntax

- Expression und Statements

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

- `struct` und `trait` als Klassen und Interfaces

```
struct MyStruct {  my_field: i32, }
impl MyStruct { fn do_smth(&self) {} }
```

- Enums

```
enum MyEnum {
  Entry1(i32, i32, i32), Entry2,
}
```

(vgl. [1, S. 122 ff.])

Hinweise zur Syntax

- Generics und `trait objects`

```
enum MyEnum<T> { Entry1(T) }
```

Hinweise zur Syntax

- Generics und `trait objects`

```
enum MyEnum<T> { Entry1(T) }
```

- Operatorenüberladung

```
impl std::ops::Add<MyEnum> for MyEnum {
  fn add(self, rhs: MyEnum) -> MyEnum { ... }
}
```

Hinweise zur Syntax

- Generics und `trait objects`

```
enum MyEnum<T> { Entry1(T) }
```

- Operatorenüberladung

```
impl std::ops::Add<MyEnum> for MyEnum {
  fn add(self, rhs: MyEnum) -> MyEnum { ... }
}
```

- Referenzen

(vgl. [10, S. 196 ff.] [7] [1, S. 246 ff.])

Vertiefung des Ownership und Borrowing

- Jeder Wert ist einer Variablen zugewiesen, dem `Owner`

Vertiefung des Ownership und Borrowing

- Jeder Wert ist einer Variablen zugewiesen, dem `Owner`
- Jeder Wert kann nur einen `Owner` besitzen

Vertiefung des Ownership und Borrowing

- Jeder Wert ist einer Variablen zugewiesen, dem `Owner`
- Jeder Wert kann nur einen `Owner` besitzen
- Verlässt der `Owner` den Gültigkeitsbereich, wird die Variable ungültig

Vertiefung des Ownership und Borrowing

- Jeder Wert ist einer Variablen zugewiesen, dem `Owner`
- Jeder Wert kann nur einen `Owner` besitzen
- Verlässt der `Owner` den Gültigkeitsbereich, wird die Variable ungültig
- Beliebige Anzahl an nicht veränderbaren Referenzen oder exakt eine veränderbare Referenz

Vertiefung des Ownership und Borrowing

- Jeder Wert ist einer Variablen zugewiesen, dem `Owner`
- Jeder Wert kann nur einen `Owner` besitzen
- Verlässt der `Owner` den Gültigkeitsbereich, wird die Variable ungültig
- Beliebige Anzahl an nicht veränderbaren Referenzen oder exakt eine veränderbare Referenz
- Referenzen müssen immer gültig sein

(vgl. [5] [6])

Beispiel Ownership

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

Beispiel 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
```

(vgl. [5] [6])

Beispiel Referenzen

```
let matrix = LargeMatrix { matrix: vec![] };

fn take_ownership(matrix_fn: LargeMatrix) { ... }
```

Beispiel Referenzen

```
let matrix = LargeMatrix { matrix: vec![] };

fn take_ownership(matrix_fn: LargeMatrix) { ... }

fn give_back_ownership(matrix_fn: LargeMatrix)
-> LargeMatrix { matrix_fn }
```

Beispiel Referenzen

```
let matrix = LargeMatrix { matrix: vec![] };

fn take_ownership(matrix_fn: LargeMatrix) { ... }

fn give_back_ownership(matrix_fn: LargeMatrix)
-> LargeMatrix { matrix_fn }

fn take_reference(matrix_ref: &LargeMatrix) { ... }
```

Beispiel Referenzen

```
let matrix = LargeMatrix { matrix: vec![] };

fn take_ownership(matrix_fn: LargeMatrix) { ... }

fn give_back_ownership(matrix_fn: LargeMatrix)
-> LargeMatrix { matrix_fn }

fn take_reference(matrix_ref: &LargeMatrix) { ... }

fn take_mutable_reference(
  matrix_ref: &mut LargeMatrix) { ... }
```

Überlauf und Ungenauigkeit

- Überlauf

```
let mut x = i32::MAX;
x += 1;
```

Überlauf und Ungenauigkeit

- Überlauf

```
let mut x = i32::MAX;
x += 1;
```

- Ungenauigkeiten

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

Überlauf und Ungenauigkeit

- Überlauf

```
let mut x = i32::MAX;
x += 1;
```

- Ungenauigkeiten

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

- Lösung mit externen Abhängigkeiten und Generics

(vgl. [4])

Überlegungen zur Datenstruktur

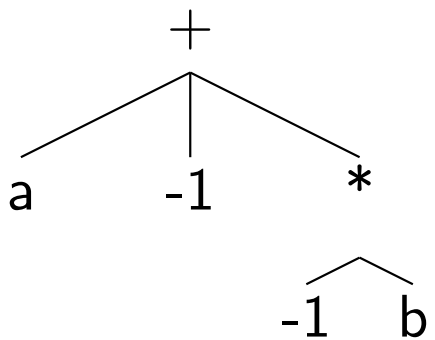
- Terme bestehen aus Token

Überlegungen zur Datenstruktur

- Terme bestehen aus Token
- Datenstruktur aus Tokens generieren

Überlegungen zur Datenstruktur

- Terme bestehen aus Token
- Datenstruktur aus Tokens generieren
- Datenstruktur als Baum



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),
}
```

Grundfunktionalitäten

- Implementierung von Operatorenüberladungen

Grundfunktionalitäten

- Implementierung von Operatorenüberladungen
- Auswertung von Termen

Erweiterung des CAS mit EvalFn

- Funktionen für Additionen und Multiplikationen

Erweiterung des CAS mit EvalFn

- Funktionen für Additionen und Multiplikationen
- Funktionen für Potenzen

Erweiterung des CAS mit EvalFn

- Funktionen für Additionen und Multiplikationen
- Funktionen für Potenzen
- Möglichkeit für mathematische Funktionen

Erweiterung des CAS mit EvalFn

- Funktionen für Additionen und Multiplikationen
- Funktionen für Potenzen
- Möglichkeit für mathematische Funktionen
- Konstantenauswertung

Erweiterung des CAS mit EvalFn

- Funktionen für Additionen und Multiplikationen
- Funktionen für Potenzen
- Möglichkeit für mathematische Funktionen
- Konstantenauswertung
- Funktionen um Terme zu vereinfachen

Vergleich zu SymPy

- Parsing von Termen identisch

Vergleich zu SymPy

- Parsing von Termen identisch
- Unterschiede sind Ausgabedingt

Vergleich zu SymPy

- Parsing von Termen identisch
- Unterschiede sind Ausgabedingt
- Bessere Performance

Verbesserungsideen und deren Ansätze

- Verbesserung der Performance

Verbesserungsideen und deren Ansätze

- Verbesserung der Performance
- Verbesserung der Erkennung von Termen

Verbesserungsideen und deren Ansätze

- Verbesserung der Performance
- Verbesserung der Erkennung von Termen
- Verbesserungen bei der Verarbeitung von Termen

Verbesserungsideen und deren Ansätze

- Verbesserung der Performance
- Verbesserung der Erkennung von Termen
- Verbesserungen bei der Verarbeitung von Termen
- Erweiterung der Substituierung

Vor- und Nachteile Rust

- Ownership- und Borrowing-System anfänglich nicht immer klar

Vor- und Nachteile Rust

- Ownership- und Borrowing-System anfänglich nicht immer klar
- Teilweise explizite Codeabschnitte

Vor- und Nachteile Rust

- Ownership- und Borrowing-System anfänglich nicht immer klar
- Teilweise explizite Codeabschnitte
- Performance und Fehlerunanfälligkeit

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

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

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

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

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

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] Rust Website: *Why Rust?*
`https://www.rust-lang.org/`.
[abgerufen am 29.08.2022].