Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht zum Proseminar
„Softwareentwicklung in der Wissenschaft“

Moderne Programmiersprachen: Rust

vorgelegt von

Christian Willner

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	7261415
Betreuer:	Michael Kuhn

Hamburg, 31. August 2020

Inhaltsverzeichnis

1. Einleitung **3**
1.1. Zweck und Einordnung 3
1.2. Nutzung 4
1.3. Anwendungsbeispiele 5

2. Syntax und Umgebung **6**
2.1. Ähnlichkeiten zu C 6
2.2. Besonderheiten der Sprache 7
2.3. Ownership 8

3. Beispiele **11**
3.1. Parallel Computing 11
3.2. Memory Safety 13

4. Zusammenfassung **15**

Literaturverzeichnis **16**

A. Vergleichs–Berechnung in C und Python (Quelltext) **18**

Abbildungsverzeichnis **20**

1. Einleitung

Was ist Rust und warum ist es so beliebt?

1.1. Zweck und Einordnung

Rust wurde konzipiert als System–Programmiersprache mit einem Fokus auf Sicherheit, Geschwindigkeit und Nebenläufigkeit. Im unten zitierten *Mission–Statement* werden außerdem integrierte Systeme, die Forderung nach verlässlichem Speicher– und Zeitverhalten und das Verhindern von Data Races als Entwicklungsziele genannt.

> Rust is a **systems programming** language focused on three goals: **safety, speed, and concurrency**. [...] making it a useful language for a number of use cases other languages aren't good at: **embedding** in other languages, programs with specific **space and time requirements**, and writing **low–level code**, like device drivers and operating systems. It improves on current languages targeting this space by having a number of **compile-time safety checks** that produce no runtime overhead, while eliminating all **data races.**
>
> – Rust Mission Statement [1]

Ursprünglich erdacht wurde Rust 2006 von Graydon Hoare. Die Entwicklung wird seit 2009 durch Mozilla gesponsort. Version 1.0 erschien 2015 und neue Updates werden alle sechs Wochen veröffentlicht. [2]
Seit 2016 belegt Rust den ersten Platz im Stack Overflow Developer Survey als beliebteste Sprache unter den mehr als 90.000 befragten Entwicklern [3]. Es ist daher von Interesse, ob Rust auch in der wissenschaftlichen Softwareentwicklung gewinnbringend eingesetzt werden kann. In diesem Sektor erfreuen sich sowohl ältere Sprachen wie Fortran, Python und C(++) großer Beliebtheit, aber auch neuere Sprachen wie Go werden immer häufiger genutzt [4]. Ein Kurzvergleich dieser Sprachen mit Rust findet sich in Tabelle 1.1.

Tabelle 1.1.: Kurzer Vergleich von Rust mit anderen Sprachen

	Rust	Python	C++	Go	Fortran
Seit	2010	1990	1985	2009	1957
Sicherheit	++	+	- -	+	-
Geschwindigkeit	++	- -	++	∘	+
Nebenläufigkeit	++	-	+	++	+
Zugang	-	++	-	∘	-

In C bzw. C++ ist es verhältnismäßig leicht Speicher fehlerhaft zu adressieren. Das liegt, wie bei Fortran, oft an der niedrigen Abstraktions–Ebene der Sprachen. Während Python und Go im normalen Betrieb gar nicht erst auf diese niedrigen Ebenen zugreifen müssen, zeigt sich für Rust ein scheinbarer Interessenkonflikt. Die Kombination von Speichersicherheit und Systemnähe ist deshalb ein Fokusthema und wird in Kapitel 2.3 vertiefend behandelt. Im direkten Geschwindigkeitsbenchmark schneidet Rust ähnlich wie C++ ab, was bei komplexen Batch–Berechnungen ein großer Vorteil ist [5]. Da sowohl Rust und Go mit einem Fokus auf Nebenläufigkeit entwickelt wurden, ist dieses Merkmal zudem relativ komfortabel nutzbar [4]. Im Allgemeinen wird Python oft als sehr zugänglich bewertet, was bei systemnahen Sprachen häufig nicht gegeben ist [6].

1.2. Nutzung

Obwohl Rust als low–level Sprache für die Systementwicklung und eingebettete Systeme konzipiert wurde, ist auffällig, dass sie auch häufig in anderen Feldern genutzt wird. Dies zeigt eine JetBrains Umfrage von 2019 (siehe Abbildung 1.1). Während der angestrebte Embedded–Bereich nur 9 % der befragten Nutzer ausmacht, zeigt die Auswertung, dass Web–Development und Spieleentwicklung ein deutlich beliebteres Anwendungsfeld für Rust sind.

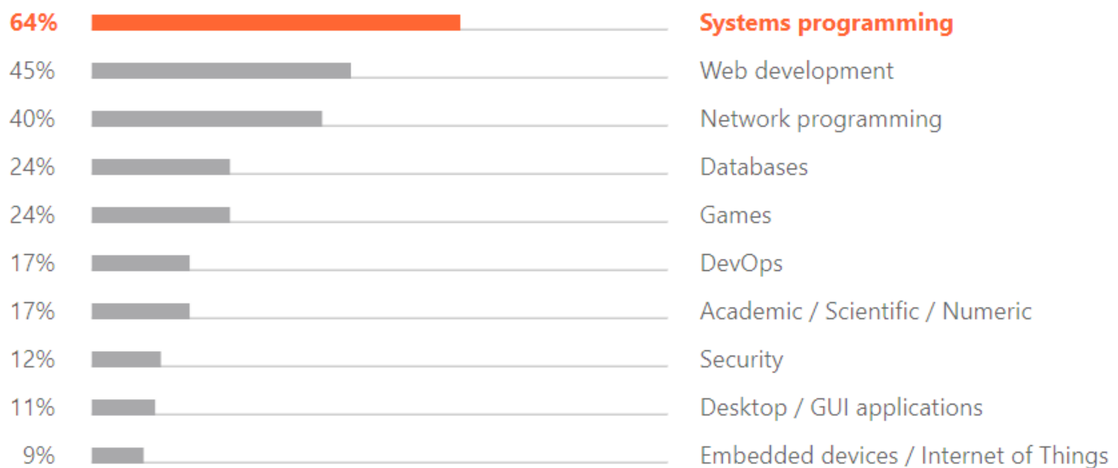

Abbildung 1.1.: Typische Anwendungsfelder von Rust [7]

1.3. Anwendungsbeispiele

Rust ist keine weitere esoterische Programmiersprache mit interessanten theoretischen Ansätzen. Als größter Unterstützer sei zunächst die Mozilla Foundation genannt. Ihr bekanntestes Projekt ist der Firefox Browser. 9,2 % der Codezeilen des Projekts sind, Stand August 2020, in Rust geschrieben. Im Mai des gleichen Jahres waren es noch 8,6 %. [8]

Die Anwendungsfelder sind divers und die Industrie ist auch außerhalb von Mozilla an der Nutzung interessiert. Dass sich die Sprache gut für die Systemprogrammierug eignet, zeigt Redox, ein unix–like Mikrokernel OS, welches vollständig in Rust geschrieben wurde [9]. Aufgrund der Sicherheitsaspekte wurde auch begonnen wichtige Funktionen des Tor–Netzwerkes in Rust abzubilden [10]. Deutlich weiter ist die Firma Dropbox, sie hat bereits den gesamten Speicherprozess auf Rust umgestellt [11]. Als weiteres Beispiel sei die Bewertungsseite Yelp genannt, sie nutzt Rust als Grundlage für A/B–testing [12].

Interessant ist auch der Wechsel der Kommunikationsplattform Discord von Go zu Rust. In diesem Anwendungsfall hatte der Garbage–Collector in Go zu regelmäßigen Einbrüchen in der Performance geführt. Nach der Portierung in Rust konnte dieses Problem behoben werden, da die Lebensdauer der Objekte schon während der Kompilierung feststeht. Ein Garbage–Collector wird nicht benötigt. [13]

2. Syntax und Umgebung

Wie sieht Rust Quelltext aus und welche Vorteile hat die Umgebung?

2.1. Ähnlichkeiten zu C

Mit einem gewissen Grundverständnis für andere Programmiersprachen, insbesondere C, lässt sich der Zweck von Rust–Code bereits gut erahnen. In Abbildung 2.1 findet sich ein typisches *Hello World* Beispiel.

Es werden zwei Funktionen definiert: `main()` und `hello()`. Die main Funktion wird bei der Ausführung des Programms aufgerufen und führt die `hello()` Funktion ohne Parameter aus. Dort wird in einer Variable `helloworld` ein String mit den Unicode–Zeichen für das japanische Guten Tag, der Name Brian und ein Emoticon gespeichert. Rust unterstützt die Eingabe von Unicode–Zeichen also bereits im Quelltext. Der Befehl `println!(...)` wird genutzt um diesen String auf der Konsole auszugeben, was im unteren Teil der Abbildung zu sehen ist. Einrückungen dienen außschließlich der Lesbarkeit. Blöcke werden über geschweifte Klammern definiert und Befehle werden mit einem Semikolon abgeschlossen.

```
fn hello() {
    let helloworld = "こんにちはBrian😻";
    println!("{}", helloworld);
}

fn main() {
    hello();
}
```

```
こんにちはBrian😻
```

Abbildung 2.1.: Hello World Beispiel mit Unicode–Zeichen

In Abbildung 2.2 werden Typdefinitionen und eine if/else–Struktur gezeigt. Der Text `let x : [u32; 3] = [47, 5, 23];` in Zeile 2 deklariert eine Variable `x` und bindet ein Array mit drei Elementen vom Typ unsigned 32–bit. Ohne die Definition des Typs, würde der Typ bei der ersten Operation mit einer anderen getypten Variable festgelegt werden oder ein entsprechender Standard gewählt (siehe Abbildung 2.3). Die if/else–Blöcke und die for–Schleife funktionieren analog zur Eingabe in C oder Java.

```rust
fn main() {
    let x : [u32; 3] = [47, 5, 23];
    let mut y : bool = true;
    for &n in &x {
        if n > 23 {
            println!("{}", n);
        }
        else {
            y = false;
        }
    }
}
```

Abbildung 2.2.: Typdefinitionen, if/else–Blöcke und for–Schleifen in Rust

2.2. Besonderheiten der Sprache

Eine Besonderheit der Sprache Rust ist der gut durchdachte Compiler mit seinen zielführenden Fehlermeldungen. Bei der Betrachtung des Codes in Abbildung 2.3 sei ein besonderes Augenmerk auf die Zeilen 2 und 4 gelegt. Der Vergleich mit `9000.0` verlangt einen Float–Typ, während `9000` ein Integer ist.

```rust
fn main() {
    let mut x = 9000.0;
    x += 1f64;
    if x > 9000 {
        println!("It's over {}!", x);
    }
}
```

Abbildung 2.3.: Fehlerhafter Rust–Code für Compiler Beispiel

Der Rust–Compiler ist bei Operationen in diesem Fall so streng, dass ein Vergleich schon zur Kompilierung nicht zugelassen wird. Das verhindert unerwünschte Effekte, die bei Vergleichen sonst häufig übersehen werden. Abbildung 2.3 zeigt die entsprechende Fehlermeldung. Die fehlerhafte Code–Stelle wird markiert und eine Lösung wird angeboten (Zeile 8).

```
error[E0308]: mismatched types
--> src/main.rs:4:12
    |
  4 |     if x > 9000 {
    |            ^^^^
    |            |
    |            expected `f64`, found integer
    |            help: use a float literal: '9000.0'

error: aborting due to previous error

For more information about this error,
try 'rustc --explain E0308'.
error: could not compile 'dragonball'.
To learn more, run the command again with --verbose.
```

Abbildung 2.4.: Compiler Error für Code aus Abbildung 2.3

Als weitere Besonderheit der Umgebung sind *Cargo* und *crates.io* zu nennen. Cargo hilft bei der automatisierten Bearbeitung von Rust Projekten. Die Unterstützung beginnt bei der Generierung der ersten Boilerplates[1], der Erstellung einer Verzeichnisstruktur und direkter Initialisierung als Git–Repository. Im Verlauf des Projekts stellt Cargo zugleich alle nötigen Werkzeuge für das Verwalten von Package–Dependencys [14], Test–Konfiguration und Dokumentations–Erstellung bereit. Im Anschluss kann Cargo dann als Package–Builder genutzt werden. [15]

Die Webseite crates.io bietet eine Schnittstelle für cargo an. Hier werden die erstellten Projekte, sogenannte *crates*, der Rust–Community für alle Nutzer frei verfügbar angeboten. Jene Registrierungsstelle bietet zum Zeitpunkt dieses Berichts (August 2020) bereits mehr als 44 000 crates an, welche insgesamt auf über 3,5 Milliarden Downloads kommen. Im Mai 2020 waren es noch 39 000 Pakete mit 2,8 Milliarden Downloads. [16]

Während diese Art der Paketverwaltungs–Plattformen für andere Sprachen schon selbstverständlich ist, war diese Eigenschaft für systemnahe Sprachen bisher eher selten. Speziell die Dependency–Verwaltung hilft dabei, dass die benutzten Pakete, inklusive aller Querverweise und deren Versionen, zentral verwaltet und wieder genutzt werden können. Eine aufwändige Abstimmung einzelner Umgebungen, wie es oft in Python der Fall ist, entfällt damit.

2.3. Ownership

In den oberen Code–Beispielen wurden einige ungewöhnliche Elemente übersprungen, die hier besprochen werden sollen. Allen voran steht die Eigenschaft, dass Variablen grundsätzlich unveränderlich sind. Wenn diese veränderbar sein sollen, muss diese Eigenschaft direkt bei der Deklaration mit dem Wort `mut` gekennzeichnet werden. In Abbildung 2.2 Zeile 3 ist dies notwendig, damit der Bool–Wert in Zeile 9 geändert werden kann.

[1] Standard Code, der immer ähnlich aussieht und häufig verwendet wird

Eine damit einhergehende, interessante Eigenschaft ist das sogenannte *Ownership* Prinzip. Es ist wie in anderen Sprachen jederzeit klar, welche Variablen auf welche Objekte zeigen. Besonders ist allerdings, dass immer nur eine Variable ein Objekt besitzen kann. Diese grundsätzliche Regelung sorgt dafür, dass *Data Races* vermieden werden können. Zur Verdeutlichung der Denkweise sei das Code–Beispiel in Abbildung 2.5 gegeben. Es wird eine Point–Struktur mit zwei Elementen in Zeile 1 definiert. In der `main` Methode wird ein Point `x` deklariert und ein Point `y`, der die gleichen Werte halten soll. Danach sollen die Werte `a` und `b` dieser Points auf der Konsole ausgegeben werden.

```
1  struct Point {a: i32, b: i32,}
2
3  fn main() {
4      let x = Point{a: 23, b: 5};
5      let y = x;
6      println!("x.a: {}, x.b: {}", x.a, x.b);
7      println!("y.a: {}, y.b: {}", y.a, y.b);
8  }
```

Abbildung 2.5.: Fehlerhafter Code, der den Ausleihprozess verdeutlicht

Der Code aus Abbildung 2.5 kann nicht kompiliert werden, da die Werte aus `x` nach `y` verschoben wurden. Damit verliert die Variable `x` praktisch alle Berechtigungen an diesem Objekt. Der Compiler schlägt dazu vor, die Point–Struktur kopierbar zu machen und beschreibt die Besitz–Veränderung über den Quellcode (vgl. Abbildung 2.6).

```
1  error[E0382]: borrow of moved value: `x`
2  --> src/main.rs:6:39
3       |
4     4 |     let x = Point{a: 23, b: 5};
5       |         - move occurs because `x` has type `Point`,
6                   which does not implement the `Copy` trait
7     5 |     let y = x;
8       |             - value moved here
9     6 |     println!("x.a: {}, x.b: {}", x.a, x.b);
10      |                                       ^^^ value borrowed
11                                                here after move
```

Abbildung 2.6.: Compiler Error für Code aus Abbildung 2.5

Eine andere Lösungsmethode ist das explizite Übergeben einer Referenz. Dazu würde Zeile 5 in Abbildung 2.5 geändert werden zu `let y = &x;`. Diese explizite Gestaltung des Ausleihens ist beim Lesezugriff von unveränderlichen Variablen zwar nicht direkt verständlich, dient aber als Grundlage für sicheres paralleles Rechnen. Das Ausleihen sorgt dafür, dass über die besitzende Variable keine Schreibzugriffe erfolgen können, solange diese für den Lesezugriff verliehen ist. Diese Beziehungs–Struktur wird während

der Programmierung bestimmt und anschließend beim Kompilieren überprüft. Damit ist auch die Lebenszeit aller Objekte definiert und es kann darauf verzichtet werden einen Garbage–Collector während der Laufzeit einzusetzen. Ein einführender Vergleich für die Performance von Rust im Parallelbetrieb ist in Kapitel 3.1 zu finden.

3. Beispiele

Inwiefern ist Rust interessant für den wissenschaftlichen Kontext?

Für die wissenschaftliche Softwareentwicklung ist es wichtig, dass große Datenmengen schnell verarbeitet werden können. Gleichzeitig ist es von Vorteil, wenn die zugehörige Sprache einfach zu verwenden ist und der eigentlichen Forschung nicht im Weg steht. Das erste Beispiel in diesem Kapitel beschäftigt sich deshalb mit der parallelen Verarbeitung großer Datensätze. Nach einem Überblick über häufige Fehlerursachen folgen darauf zwei Beispiele, wie Rust mit den begrenzten Zahlenräumen der Computerarithmetik umgeht, um zu zeigen, wie Fehler früh aufgedeckt werden können. Für die Vermeidung von Fehlern ist der Compiler ein sehr hilfreiches Tool. Beispiele dafür wurden bereits in Kapitel 2 vorgestellt.

3.1. Parallel Computing

Bei der Verarbeitung großer Datenmengen ist es häufig möglich Teilstrukturen unabhängig voneinander zu berechnen, weil die Ergebnisse nicht voneinander abhängen. Diese abgeschlossenen Teile können dann parallel berechnet werden, um die Gesamtlaufzeit zu reduzieren. Der Programmcode in Abbildung 3.1 soll eine solche Prozedur nachempfinden. Es werden zwei Vektoren mit gleichem Inhalt erstellt (Zeile 7–9). Jeder Vektor hat 10 Millionen Elemente, für die jeweils parallel bzw. seriell eine geometrische Rechnung durchgeführt wird (Z. 10–13).

```rust
extern crate rayon;
use rayon::prelude::*;
extern crate rand;
use rand::Rng;
fn main() {
  const VEC_SIZE: usize = 10_000_000;
  let mut vec1: Vec<f64> = (0..VEC_SIZE)
    .map(|_| rng.gen::<f64>()).collect();
  let mut vec2 = vec1.clone();
  vec1.iter_mut()
    .for_each(|p| *p *= p.sin() * p.cos()); // serial
  vec2.par_iter_mut()
    .for_each(|p| *p *= p.sin() * p.cos()); // parallel
}
```

Abbildung 3.1.: Beispiel für parallele Verarbeitung von großen Datenmengen in Rust

Rust ist im Kern sehr minimal ausgelegt und viele integrierte Funktionen anderer Programmiersprachen werden in optionale Pakete ausgelagert. Im aktuellen Beispiel werden Zufallszahlen generiert, weshalb in Zeile 3 das rand Paket importiert werden muss. Die darauffolgende Zeile ermöglich das Aufrufen der `rng.gen` Funktion ohne Präfix. Ohne Zeile 4 müsste der Funktionsaufruf als `rand::Rng::rng.gen` erfolgen. In den ersten beiden Zeilen des Programmcodes wird auf die gleiche Weise das rayon Paket eingebunden. Rayon ist im Moment das am weitesten verbreitete Paket für parallele Berechnungen. Die einfache Umstellung der bisherigen seriellen Berechnung gegen die parallele Version ist in den Zeilen 11 und 13 zu sehen: `iter_mut()` wird durch `par_iter_mut()` ersetzt.

Diese Berechnung wurde für eine kurze Untersuchung ebenfalls in C und Python implementiert. Die Parallelität wurde in C über OpenMP erreicht. Python wurde nur seriell implementiert. Der Quellcode für C und Python ist im Anhang gelistet (vgl. Abbildungen A.1 und A.2). Alle Versionen wurden auf dem gleichen System berechnet. Das C–Programm wurde mit OpenMP über Windows Subystem for Linux ausgeführt, was etwaige Geschwindigkeitseinbußen erklären könnte. Die Setup–Phase wurde ausgeklammert und nur die eigentliche Berechnungszeit bestimmt. In Tabelle 3.2 sind die gemittelten Zeiten über 10 Ausführungen gelistet.

Sprache	Seriell	Parallel
Rust	0.483	0.196
C	1.114	0.432
Python	16.8	-

Abbildung 3.2.: Ergebnis der Messung aus Abbildung 3.1 in Sekunden

Die Ausführungszeit des C Programms wurde nach bestem Wissen optimiert, die Konfiguration von OpenMP lässt hier viele Möglichkeiten zu. Während der Kompilierung

wurden ebenfalls verschiedene Optimierungslevel ausprobiert. Die schnellste Kombination wurde übernommen. Es ist nicht auszuschließen, dass der Code bei besserer Konfiguration ähnlich schnell rechnet wie die Rust Version. Unabhängig davon ist in jedem Fall eine Größenordnung für die Einordnung erkennbar. Erstaunlich ist, dass diese Messwerte ohne Anpassung des Standard Rust–Codes für das rayon Paket entstanden sind. Die Benutzung ist ausgehend von diesem Beispiel nicht nur schnell, sondern auch einfach.

3.2. Memory Safety

Betrachtet man die häufigsten Fehlerquellen in der Programmierung stellvertretend anhand der Linux CVE[1] von 2018 ergibt sich die Verteilung aus Abbildung 3.3. Viele dieser Fehler wären in Rust bereits bei der Kompilierung aufgefallen, bzw. wären *use after free* und *double free* in Rust nicht möglich. Die Lebenszeit wird durch den Compiler anhand des Scopes oder von Lifetime–Attributen festgelegt. Sie ist von der Programmierung größtenteils abgekapselt.

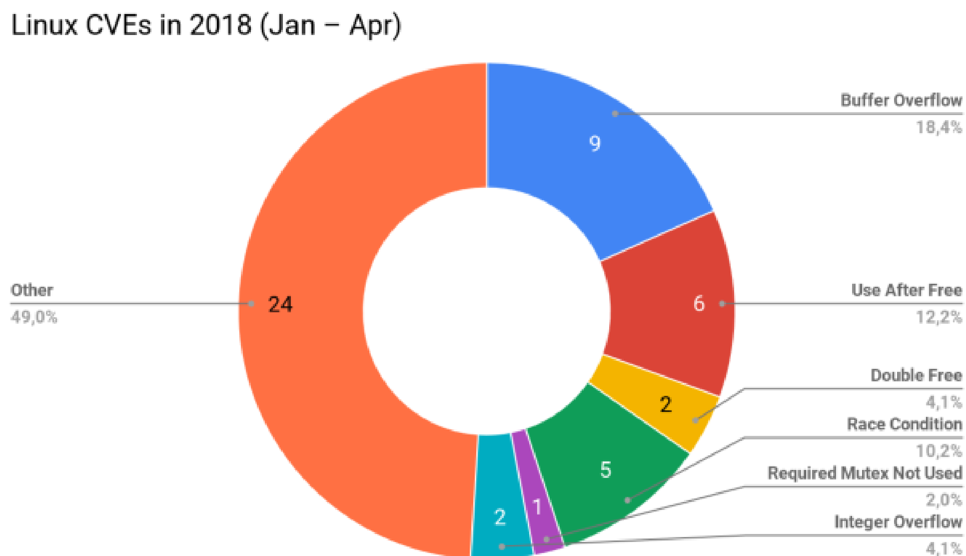

Abbildung 3.3.: Häufigste Linux Fehlerquellen 2018 [17]

Am Beispiel des *Integer Overflow* lassen sich zwei Verhaltensarten zeigen. Wenn während der Kompilierung bereits klar ist, dass es zu einem Overflow kommt, wird eine Fehlermeldung ähnlich zu Abbildung 3.4 ausgegeben. Für den Fall eines Overflows während der Ausführung, kommt es zu einem Panik–Abbruch wie in Abbildung 3.5. Der Nutzer kann dadurch sicher sein, dass es im bisherigen Ablauf nicht zu einem Overflow gekommen ist.

[1] Common Vulnerabilities and Exposures

```
error: literal out of range for `i8`
--> src/main.rs:3:23
    |
  3 |     let _y: i8 = &x + 256;
    |                       ^^^
    |
    = note: `#[deny(overflowing_literals)]` on by default

error: aborting due to previous error
```

Abbildung 3.4.: Compiler Error für einen Integer Overflow zur Kompilierzeit

```
    thread 'main' panicked at 'attempt to add
    with overflow', .\overflow.rs:12:18
    note: run with `RUST_BACKTRACE=1` environment
    variable to display a backtrack
```

Abbildung 3.5.: Abbruch nach einem Integer Overflow zur Laufzeit

4. Zusammenfassung

Das Kernargumente für Rust ist die Kombination von Geschwindigkeit, Sicherheit und Nebenläufigkeit. Zwar ist die Stellung von Rust damit nicht von heute auf morgen marktbeherrschend, doch erfreut sich die Sprache steigender Beliebtheit.

> Rust promises performance, control, memory safety, and fearless concurrency – an enticing combination, especially for systems programming. [. . .] Coupled with an open development process, it makes sense that many programmers (even those that don't use it) hold Rust in high esteem.
>
> – The Overflow [18]

Das Anwendungsgebiet war anfangs sehr auf systemnahe Umgebungen spezialisiert, die große Community entwickelte allerdings Pakete für ein deutlich größeres Feld von Anwendungen. Die besonders transparente Art der Entwicklung und der Enthusiasmus der Unterstützer führten zu einer ungewöhnlich zugänglichen low–level Sprache. Im August 2020 zählt das Github–Repository von Rust mehr als 2 900 Contributors, das Cargo System wurde von mehr als 650 Menschen bearbeitet. [19]

Insgesamt sind sich viele Entwickler einig, dass die Ziele einer sicheren Entwicklung mit Rust gut erreichbar sind und unkompliziert umgesetzt werden können [18]. Rust eignet sich aus diesen Gründen hervorragend für den Einsatz in der wissenschaftlichen Softwareentwicklung.

Literaturverzeichnis

[1] “The Rust Programming Language – Readme,” https://doc.rust-lang.org/1.1.0/book/README.html Stand: 2020-08-06.

[2] S. Klabnik, “History of Rust,” 2016, http://steveklabnik.github.io/history-of-rust/ Stand: 2020-08-06.

[3] “The 2019 Stack Overflow Developer Survey Results Are In,” 2019, https://stackoverflow.blog/2019/04/09/the-2019-stack-overflow-developer-survey-results-are-in/ Stand: 2020-08-06.

[4] J. Fernández-Villaverde, P. Guerrón, and D. Z. Valencia, “Scientific Computing Languages,” 2019, https://www.sas.upenn.edu/~jesusfv/Lecture_HPC_5_Scientific_Computing_Languages.pdf Stand: 2020-08-06.

[5] “Which programs are fastest?” https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html Stand: 2020-08-06).

[6] D. Swersky, “Top 43 Programming Languages: When and How to Use Them,” 2018, https://raygun.com/blog/programming-languages/ Stand: 2020-08-06.

[7] “Rust,” 2019, https://www.jetbrains.com/lp/devecosystem-2019/rust/ Stand: 2020-08-06.

[8] D. Bushev, “How much Rust in Firefox?” 2020, https://4e6.github.io/firefox-lang-stats/ Stand: 2020-08-07.

[9] “Redox OS landing page,” 2020, https://redox-os.org/ Stand: 2020-08-07.

[10] S. Hahn, “[tor-dev] Tor in a safer language: Network team update from Amsterdam,” 2017, https://lists.torproject.org/pipermail/tor-dev/2017-March/012088.html Stand: 2020-08-07.

[11] C. Metz, “The Epic Story of Dropbox’s Exodus From the Amazon Cloud Empire,” 2016, https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/ Stand: 2020-08-07.

[12] A. Verardi and F. Raynaud, “How to write Rust instead of C, and get away with it (yes, it’s a Python talk),” 2018, https://www.youtube.com/watch?v=u6ZbF4apABk Stand: 2020-08-07.

[13] J. Howarth, “Why Discord is switching from Go to Rust,” 2020, https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f Stand: 2020-08-07.

[14] “Cargo – The Rust package manager (Github–Repository,” 2020, https://github.com/rust-lang/cargo/ Stand: 2020-08-07.

[15] “The Cargo Book,” 2020, https://doc.rust-lang.org/cargo/ Stand: 2020-08-10.

[16] “The Rust community’s crate registry,” 2020, https://crates.io/ Stand: 2020-08-10.

[17] P. Oppermann, “The Rust Way of OS Development,” 2018, https://phil-opp.github.io/talk-konstanz-may-2018/ Stand: 2020-08-10.

[18] “The 2020 Developer Survey results are here!” 2020, https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/ Stand: 2020-08-10.

[19] “The Rust Programming Language ,” 2020, https://github.com/rust-lang Stand: 2020-08-07.

A. Vergleichs–Berechnung in C und Python (Quelltext)

```python
import random
import math
import time
vec_size = 10000000
arr = [random.random() for _ in range(vec_size)]
start_time = time.time()
for p in arr:
    p *= math.sin(p) * math.cos(p) - math.acos(p)
end_time = time.time()
print("Serial: {:.5f} seconds".format(end_time - start_time))
```

Abbildung A.1.: Vergleichs–Berechnung in Python (nur seriell)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>

int main(){
    srand(time(0));
    int VEC_SIZE = 10000000;
    double *arr = (double*)malloc(VEC_SIZE * sizeof(double));
    double *arr2 = (double*)malloc(VEC_SIZE * sizeof(double));
    int i;
    for (i = 0; i < VEC_SIZE; ++i){
    arr[i] = (double)rand()/(double)(RAND_MAX);
    arr2[i] = arr[i];
    }
    struct timespec start, finish;

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (i = 0; i < VEC_SIZE; ++i){
    arr[i] = sin(arr[i]) * cos(arr[i]) - acos(arr[i]);
    }
    clock_gettime(CLOCK_MONOTONIC, &finish);
    double time_taken;
    time_taken = (finish.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (finish.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("C Single thread: %f seconds\n", time_taken);

    clock_gettime(CLOCK_MONOTONIC, &start);
    #pragma omp parallel
    {
        #pragma omp for schedule(static,VEC_SIZE/4)
        for (i = 0; i < VEC_SIZE; ++i){
            arr2[i] = sin(arr2[i]) * cos(arr2[i]) - acos(arr2[i]);
        }
    }
    clock_gettime(CLOCK_MONOTONIC, &finish);

    time_taken = (finish.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (finish.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("C Parallel thread: %f seconds\n", time_taken);

    return 0;
}
```

Abbildung A.2.: Vergleichs–Berechnung in C

Abbildungsverzeichnis

1.1. Typische Anwendungsfelder von Rust [7] 5

2.1. Hello World Beispiel mit Unicode–Zeichen 6
2.2. Typdefinitionen, if/else–Blöcke und for–Schleifen in Rust 7
2.3. Fehlerhafter Rust–Code für Compiler Beispiel 7
2.4. Compiler Error für Code aus Abbildung 2.3 8
2.5. Fehlerhafter Code, der den Ausleihprozess verdeutlicht 9
2.6. Compiler Error für Code aus Abbildung 2.5 9

3.1. Beispiel für parallele Verarbeitung von großen Datenmengen in Rust . . . 12
3.2. Ergebnis der Messung aus Abbildung 3.1 in Sekunden 12
3.3. Häufigste Linux Fehlerquellen 2018 [17] 13
3.4. Compiler Error für einen Integer Overflow zur Kompilierzeit 14
3.5. Abbruch nach einem Integer Overflow zur Laufzeit 14

A.1. Vergleichs–Berechnung in Python (nur seriell) 18
A.2. Vergleichs–Berechnung in C 19