

Fakultät Informatik

Sicherheitsaspekte Rust

Bericht im Studiengang Informatik

vorgelegt von
Robin Rosner

Matrikelnummer 362 5303

© 2024

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

Abstract

Dieser Bericht behandelt die zentralen Sicherheitsaspekte der Programmiersprache Rust, die entwickelt wurde, um sichere und effiziente Software zu schreiben. Rust kombiniert leistungsstarke Kontrolle über Systemressourcen mit modernen Spracheigenschaften, die viele häufige Sicherheitsprobleme in der Softwareentwicklung verhindern [1].

Zunächst wird die statische Typisierung von Rust erläutert, die durch strikte Typenkontrolle zur Vermeidung typischer Fehler beiträgt. Anschließend wird gezeigt, welche Möglichkeiten Rust bietet mit Integer-Überläufe umzugehen. Ein weiterer Schwerpunkt liegt auf den Zero-Cost-Abstraktionen, die es Entwicklern ermöglichen, effizienten und dennoch sicheren Code zu schreiben.

Der Bericht beleuchtet auch Rusts Ansätze zur Fehlerbehandlung, die auf das Konzept der Ausnahmen verzichten und stattdessen explizite Fehlerwerte nutzen. Besonders hervorzuheben ist Rusts Besitz- und Ausleihmodell, das durch strikte Regeln für Speicherzugriffe und Lebensdauern von Objekten die Sicherheit und Stabilität des Codes gewährleistet. Ergänzend dazu wird das Lebensdauer-Tracking vorgestellt, das durch präzise Kontrolle über die Lebensdauer von Referenzen die Gefahr von hängenden Referenzen minimiert.

Schließlich wird Rusts Ansatz zur sicheren Nebenläufigkeit untersucht, der Datenrennen (Datenrennens) und damit verbundene Sicherheitsprobleme verhindert. Durch diese Sicherheitsmechanismen bietet Rust eine robuste Basis für die Entwicklung sicherer und zuverlässiger Software, die den Anforderungen moderner Systemprogrammierung gerecht wird.

Dieser Bericht gibt einen Überblick über die wichtigsten Spracheigenschaften von Rust und zeigt anhand Beispiele, wie diese in der Praxis angewendet werden können.

Inhaltsverzeichnis

1 Statische Typen	1
1.1 Theoretische Grundlagen	1
1.2 Praktische Beispiele	1
1.2.1 Vergleich mit anderen Programmiersprachen	2
1.2.2 Sicherheitsvorteile der statischen Typen in Rust	2
2 Integer-Überlauf	3
2.1 Theoretische Grundlagen	3
2.2 Praktische Beispiele	3
2.3 Vergleich mit anderen Programmiersprachen	4
2.4 Sicherheitsvorteile der Rust-Features	4
3 Zero-Cost-Abstraktionen	5
3.1 Theoretische Grundlagen	5
3.2 Praktische Beispiele	5
3.2.1 Iteratoren	5
3.2.2 Message Passing	6
3.2.3 Option Typ	6
3.2.4 Filter-Funktion	7
3.3 Vergleich mit anderen Programmiersprachen	7
3.4 Sicherheitsvorteile der Zero-Cost-Abstraktionen	7
4 Fehlerbehandlung	8
4.1 Theoretische Grundlagen	8
4.2 Praktische Beispiele	8
4.3 Vergleich mit anderen Programmiersprachen	9
4.4 Sicherheitsvorteile	9
5 Besitz und Ausleihe	10
5.1 Theoretische Grundlagen	10
5.2 Praktische Beispiele	10
5.2.1 Copy-Typen	10
5.2.2 Ausleihe	11
5.2.3 Veränderliche Referenzen	11
5.2.4 Kopieren und Klonen	12
5.3 Vergleiche zu anderen Programmiersprachen	12
5.4 Sicherheitsvorteile der Besitz- und Ausleihregeln	13
6 Lebensdauer-Tracking	14
6.1 Theoretische Grundlagen	14
6.2 Praktische Beispiele	14
6.3 Vergleich mit anderen Programmiersprachen	15
6.3.1 Sicherheitsvorteile	15
6.4 Fazit	15

7 Sichere Nebenläufigkeit	16
7.1 Theoretische Grundlagen	16
7.2 Praktische Beispiele	17
7.3 Vergleich mit anderen Programmiersprachen	18
7.4 Sicherheitsvorteile	18
8 Fazit	19
Beispielverzeichnis	20
Literaturverzeichnis	21
Glossar	22

1 Statische Typen

1.1 Theoretische Grundlagen

Rust ist eine statisch typisierte Programmiersprache, was bedeutet, dass die Typen aller Variablen zur Kompilierzeit bekannt sein müssen. Dies ermöglicht dem Compiler, viele Fehler frühzeitig zu erkennen und zu verhindern, dass unsichere Operationen wie Dereferenzierungen von Nullzeigern oder ungesicherte Typumwandlungen, Typfehler und Typinkompatibilitäten ausgeführt werden.

1.2 Praktische Beispiele

Listing 1.1: i32 in i64

```
1 pub fn i32_in_i64() {
2     let smol: i32 = 128;
3
4     // panic: i32 will not get converted to i64
5     let big: i64 = smol;
6
7     // explicit conversion from i32 to i64
8     let big: i64 = smol.into();
9
10    println!("i32_in_i64: {}", big);
11 }
```

In diesem Beispiel wird versucht, einen `i32`-Wert direkt in einen `i64`-Wert zu konvertieren, was nicht erlaubt ist. Stattdessen kann die Methode `into()` verwendet werden, um explizit eine Typenkonvertierung zu veranlassen. Rust führt keine impliziten Konvertierungen durch; der Programmierer muss an dieser Stelle explizit sein [2, Kapitel 1. Types].

Listing 1.2: Signiertheit

```
1 pub fn signdness() {
2     let unsigned_val: u32 = 150;
3     let signed_val: i32 = -100;
4
5     // will panic
6     let signed: i32 = unsigned_val;
7     let unsigned: u32 = signed_val;
8
9     // explicit conversion from unsigned to signed
10    let signed_from_unsigned: i32 = unsigned_val as i32;
11    println!("Signed_from_Unsigned: {}", signed_from_unsigned);
12
13    // explicit conversion from signed to unsigned
14    // has a logical error in this context and overflows
15    let unsigned_from_signed: u32 = signed_val as u32;
16    println!("Unsigned_from_Signed: {}", unsigned_from_signed);
17 }
```

In diesem Beispiel zeigt die Funktion `signedness()` die möglichen Probleme bei der Konvertierung zwischen signierten und unsignierten Typen. Direkte Konvertierungen können zu logischen Fehlern oder Überläufen führen, daher ist es wichtig, explizite und sichere Methoden zu verwenden. Auch hier muss der Programmierer explizit eine Konvertierung angeben.

1.2.1 Vergleich mit anderen Programmiersprachen

In vielen dynamisch typisierten Sprachen wie Python oder JavaScript werden Typfehler erst zur Laufzeit entdeckt, was zu unvorhersehbarem Verhalten und potenziellen Sicherheitslücken führen kann.

1.2.2 Sicherheitsvorteile der statischen Typen in Rust

Die statische Typisierung in Rust bietet mehrere Sicherheitsvorteile:

- **Frühe Fehlererkennung:** Typfehler werden während der Kompilierung erkannt, was die Wahrscheinlichkeit von Laufzeitfehlern verringert [3].
- **Explizite Konvertierungen:** Automatische und implizite Typkonvertierungen werden in Rust vermieden, was dazu beiträgt, unerwartete Fehler zu vermeiden [2, Kapitel 1. Types].
- **Strikte Typüberprüfung:** Der Compiler erzwingt strenge Regeln für Typen und ihre Verwendung, was zu sichererem und stabilerem Code führt [3].

2 Integer-Überlauf

2.1 Theoretische Grundlagen

Der Integer-Überlauf ist ein häufiges Problem in vielen Programmiersprachen, das zu unerwartetem Verhalten und Sicherheitslücken führen kann. Ein Integer-Überlauf tritt auf, wenn eine arithmetische Operation das Maximum (oder Minimum) des für den Datentyp zulässigen Wertebereichs überschreitet. In vielen Programmiersprachen führt dies zu undefiniertem Verhalten oder unvorhersehbaren Ergebnissen, was potenziell ausnutzbare Schwachstellen zur Folge haben kann [4].

Rust hingegen bietet verschiedene Mechanismen, um Integer-Überläufe sicher zu handhaben und das Verhalten explizit zu definieren [5, Kapitel 3.2]. Diese Mechanismen sorgen dafür, dass der Code sicherer und robuster wird.

2.2 Praktische Beispiele

Das folgende Beispiel zeigt, wie Rust Integer-Überläufe in verschiedenen Szenarien behandelt:

Listing 2.1: Integer-Überläufe

```
1 pub fn overflow_how_not() {
2     let a: i32 = i32::MAX;
3     // panic in release mode due to runtime checks
4     // wil wrap in release mode -> defined behaviour
5     let b = a + 1;
6
7     println!("{}", b);
8 }
9 pub fn overflow() {
10     let a: i32 = i32::MAX;
11
12     // checked addition
13     let checked_sum = a.checked_add(1);
14     println!("Checked_sum: {}", checked_sum);
15
16     // wrapping addition
17     let wrapping_sum = a.wrapping_add(1);
18     println!("Wrapping_sum: {}", wrapping_sum);
19
20     // saturating addition
21     let saturating_sum = a.saturating_add(1);
22     println!("Saturating_sum: {}", saturating_sum);
23
24     // overflowing addition
25     let (overflowing_sum, overflowed) = a.overflowing_add(1);
26     println!("Overflowing_sum: {}, overflowed: {}",
27             overflowing_sum, overflowed);
28 }
```

2.3 Vergleich mit anderen Programmiersprachen

In vielen Programmiersprachen wie C und C++ wird ein Integer-Überlauf nicht automatisch erkannt und ein Überlauf resultiert in undefiniertem Verhalten. Dies kann schwerwiegende Sicherheitslücken verursachen, die von Angreifern ausgenutzt werden können. Rust überprüft auf Überläufe im Debug-modus. Im Release-modus ist diese Überprüfung aus performance Gründen standardmäßig deaktiviert. Ein Über- oder Unterlauf ist in Rust jedoch immer definiertes Verhalten mit einem *Wrap-around*.

Rust bietet mehrere Möglichkeiten, Integer-Überläufe sicher zu handhaben:

- **Überprüfte Addition (checked addition):** Gibt `None` zurück, wenn ein Überlauf auftritt.
- **Wrappende Addition (wrapping addition):** Wickelt den Wert bei Überlauf um, entsprechend dem Wertebereich des Datentyps.
- **Sättigende Addition (saturating addition):** Begrenzt den Wert auf den maximalen oder minimalen Wert des Datentyps bei Überlauf.
- **Überlaufende Addition (overflowing addition):** Gibt ein *Tupel* zurück, bestehend aus dem Ergebnis der Addition und einem booleschen Wert, der anzeigt, ob ein Überlauf aufgetreten ist.

Diese Mechanismen stellen sicher, dass der Entwickler das Verhalten bei einem Überlauf explizit kontrollieren und somit sichereren Code schreiben kann.

2.4 Sicherheitsvorteile der Rust-Features

Durch definiertes Verhalten und die Möglichkeit für explizite Handhabung von Integer-Überläufen, wie in Beispiel 2.1 gezeigt, trägt Rust erheblich zur Sicherheit von Anwendungen bei. Entwickler können bewusst entscheiden, wie Überläufe behandelt werden sollen, und vermeiden so unvorhersehbare und potenziell gefährliche Situationen. Die verschiedenen Methoden zur Überlaufbehandlung ermöglichen es, die Anforderungen unterschiedlicher Anwendungsfälle zu erfüllen, ohne Kompromisse bei der Sicherheit einzugehen.

Zusammengefasst bietet Rust durch seine strikte und transparente Handhabung von Integer-Überläufen eine solide Basis für die Entwicklung sicherer Software, die weniger anfällig für typische Sicherheitslücken ist, wie sie in anderen Programmiersprachen häufig auftreten [6].

3 Zero-Cost-Abstraktionen

Zero-Cost-Abstraktionen sind ein zentrales Konzept in Rust, das es ermöglicht, hohe Abstraktionen zu verwenden, ohne die Laufzeit des Programms zu beeinträchtigen. Dies bedeutet, dass der Overhead, der durch die Abstraktionen entsteht, zur Kompilierzeit eliminiert wird, sodass der generierte Maschinencode ähnlich effizient ist wie handgeschriebener Low-Level-Code [7].

3.1 Theoretische Grundlagen

In vielen Programmiersprachen führen Abstraktionen zu einem gewissen Laufzeit-Overhead, sei es durch zusätzliche Funktionsaufrufe, Speicherallokationen oder andere Mechanismen. Rusts Zero-Cost-Abstraktionen gewährleisten, dass solche Abstraktionen zur Kompilierzeit aufgelöst werden, wodurch der generierte Code ähnlich speicher- und laufzeiteffizient bleibt wie bei direkter Implementierung [8].

3.2 Praktische Beispiele

Hier sind einige Beispiele für Zero-Cost-Abstraktionen in Rust:

- Iteratoren
- Nachrichtenübermittlung
- Der Option-Typ
- Filter-Funktion

3.2.1 Iteratoren

Rust bietet leistungsstarke Iteratoren, die keine zusätzlichen Laufzeitkosten verursachen. Hier ist ein Beispiel, das zeigt, wie man sicher durch einen Vektor iteriert:

Listing 3.1: Iterator

```
1 use std::sync::mpsc;
2 use std::thread;
3
4 pub fn iterator(){
5     let numbers = vec![1, 2, 3, 4, 5];
6
7     // Using an iterator to iterate through the vector safely
8     for num in numbers.iter() {
9         println!("{}", num); // Safely prints each number
10    }
11 }
```

Ein solcher Iterator löst die Programmlogik von der Schleifenlogik, wodurch typische Fehler wie *off-by-one* und daraus resultierende falsche Array-Indexierung vermieden werden.

3.2.2 Message Passing

Message Passing ist eine sichere und effiziente Methode zur Kommunikation zwischen Threads, die in Rust ebenfalls ohne Laufzeitkosten implementiert ist:

Listing 3.2: Message Passing

```

1 pub fn message_passing() {
2     let (tx, rx) = mpsc::channel();
3
4     let sender_thread = thread::spawn(move || {
5         let msg = "Hello_from_the_sender_thread";
6         // Sends a message safely to the receiver
7         tx.send(msg).unwrap();
8         println!("Sent_message: '{}'", msg);
9     });
10
11    let receiver_thread = thread::spawn(move || {
12        // Receives message safely
13        let received = rx.recv().unwrap();
14        println!("Received_message: '{}'", received);
15    });
16
17    sender_thread.join().unwrap();
18    receiver_thread.join().unwrap();
19 }

```

Durch eine solche einfach Abstraktion können Datenrennen zwischen threads vermieden werden. Trotzdem können Threads effektiv kommunizieren und der Code bleibt lesbar.

3.2.3 Option Typ

Der `Option` Typ wird verwendet, um mögliche Abwesenheit eines Wertes sicher zu handhaben. Hier ist ein Beispiel, wie man eine Division sicher durchführen kann:

Listing 3.3: Option Typ

```

1 pub fn option_type() {
2     let dividend = 10;
3     let divisor = 0;
4
5     match find_divisor(dividend, divisor) {
6         Some(result) => println!("Result_of_division: {}", result),
7         None => println!("Cannot_divide_by_zero!"),
8     }
9 }
10
11 fn find_divisor(dividend: i32, divisor: i32) -> Option<i32> {
12     if divisor == 0 {
13         None // Safely handle division by zero
14     } else {
15         Some(dividend / divisor) // Safely return the result
16     }
17 }

```

Eine `Option` gibt entweder einen wert `Some` zurück oder einen `None` Wert. Mittels `match` kann geprüft werden ob ein Wert vorliegt und dementsprechend verarbeitet werden.

3.2.4 Filter-Funktion

Die `filter` Methode ermöglicht es, Elemente eines Iterators basierend auf einer Bedingung zu filtern, ohne zusätzlichen Laufzeitaufwand:

Listing 3.4: Filter-Funktion

```
1 pub fn filter() {  
2     let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
3  
4     // Use filter to find even numbers  
5     let even_numbers: Vec<_> = numbers.iter()  
6         .filter(|&x| x % 2 == 0)  
7         .collect();  
8  
9     println!("Even numbers: {:?}", even_numbers);  
10 }
```

Ein Filter auf einem Iterator separiert zusätzlich die Filter logik. An dieser stelle könnte auch eine Methode verwendet werden. Dies sorgt für besser lesbaren Code und beugt logik fehlern vor (off by one etc.).

3.3 Vergleich mit anderen Programmiersprachen

In vielen anderen Programmiersprachen, wie zum Beispiel Python oder Java, führen hohe Abstraktionen häufig zu einem gewissen Maß an Laufzeitkosten. Diese können in Form von zusätzlichen Funktionsaufrufen, Garbage Collection oder Speicherallokationen auftreten. Rust eliminiert diese Kosten, indem es die Abstraktionen zur Kompilierzeit auflöst und so effizienten Maschinencode generiert.

3.4 Sicherheitsvorteile der Zero-Cost-Abstraktionen

Die Verwendung von Zero-Cost-Abstraktionen in Rust trägt zur Sicherheit des Codes bei. Vorimplementierte abstraktionen sind gut getestet, machen den Code lesbarer, abstrahieren z.B. die iteration und verhindern so *off-by-one* Fehler.

4 Fehlerbehandlung

Rust bietet einen robusten Ansatz zur Fehlerbehandlung, der ohne Ausnahmen auskommt und eine sichere und vorhersehbare Fehlerbehandlung ermöglicht. Im Gegensatz zu vielen anderen Programmiersprachen verwendet Rust das `Result` und `Option` Typsystem, um Fehler und fehlende Werte explizit zu behandeln.

4.1 Theoretische Grundlagen

In Rust werden Fehler durch die Typen `Result<T, E>` und `Option<T>` behandelt. `Result<T, E>` ist ein Typ, der entweder einen Wert vom Typ `T` oder einen Fehler vom Typ `E` enthält.

4.2 Praktische Beispiele

Im Folgenden werden zwei Beispiele gezeigt: eines, das keine sichere Fehlerbehandlung verwendet, und eines, das Rusts robusten Ansatz zur Fehlerbehandlung nutzt.

Listing 4.1: Unsichere Fehlerbehandlung

```
1 use std::fs::File;
2 use std::io::Read;
3
4 pub fn get_file_unsave() {
5     // This will panic if there's an error
6     let contents = read_file_content_unsave();
7     println!("File_{}_contents:{}", contents);
8 }
9
10 fn read_file_content_unsave() -> String {
11     // Risky if file is not found
12     let mut file = File::open("example.txt").unwrap();
13     let mut contents = String::new();
14     // Panics on error
15     file.read_to_string(&mut contents).unwrap();
16     contents
17 }
```

Listing 4.2: Sichere Fehlerbehandlung

```

1 use std::fs::File;
2 use std::io::Read;
3
4 pub fn get_file() {
5     match read_file_content() {
6         Ok(contents) => println!("File contents: {}", contents),
7         Err(e) => println!("Error reading file: {}", e),
8     }
9 }
10
11 fn read_file_content() -> Result<String, std::io::Error> {
12     let mut file = File::open("example.txt");
13     let mut contents = String::new();
14     file.read_to_string(&mut contents)?;
15     Ok(contents)
16 }

```

Im ersten Beispiel wird keine Fehlerbehandlung vorgenommen. Das Programm wird mit `unwrap()` gezwungen, den Inhalt der Datei zu lesen, was zum Absturz des Programms führt, falls ein Fehler auftritt. Im zweiten Beispiel wird die `Result`-Typ verwendet, um Fehler abzufangen und zu behandeln, was zu einem robusteren und sichereren Code führt.

`Unwrap()` gibt von einem Fehlertyp den Wert zurück falls ein Fehler vorliegt geht das Programm in eine *panic*. Mittels `match` kann überprüft werden, ob das `Result` OK ist und ein Wert vorliegt oder ob das `Result` ein Fehler ist.

4.3 Vergleich mit anderen Programmiersprachen

In vielen anderen Programmiersprachen, wie z.B. C++ oder Java, werden Ausnahmen (*exceptions*) verwendet, um Fehler zu behandeln. Rusts Ansatz, Fehler explizit zu behandeln, führt zu einem vorhersehbaren Verhalten und fördert den bewussten Umgang mit Fehlern [5, Kapitel 9]. Es müssen alle möglichen Fehler behandelt werden, auch wenn dies nur mittels `unwrap()` passiert.

- **C++:** Verwendet Ausnahmen, die oft schwer nachzuvollziehen und zu debuggen sind.
- **Java:** Nutzt ebenfalls Ausnahmen, wobei der Entwickler sicherstellen muss, dass alle möglichen Ausnahmen abgefangen werden.
- **Rust:** Verwendet das `Result` und `Option` Typsystem, um Fehler explizit zu behandeln.

4.4 Sicherheitsvorteile

Rusts Ansatz zur Fehlerbehandlung bietet mehrere Sicherheitsvorteile:

- **Vorhersehbarkeit:** Da Fehler explizit behandelt werden müssen, ist das Verhalten des Programms vorhersehbarer.
- **Vermeidung von Panics:** Durch die Verwendung von `Result` und `Option` können viele *panics* vermieden werden, was die Stabilität des Programms erhöht.
- **Robustheit:** Der Code wird robuster, da alle möglichen Fehlerfälle berücksichtigt und behandelt werden, sofern das unsichere `unwrap()` vermieden wird.

5 Besitz und Ausleihe

5.1 Theoretische Grundlagen

In Rust sind Besitz und Ausleihe grundlegende Konzepte, die zur Gewährleistung der Speicher- und Datensicherheit beitragen. Das Besitzmodell sorgt dafür, dass es stets einen eindeutigen Besitzer eines Datenobjekts gibt, während das Ausleihmodell ermöglicht, dass andere Teile des Codes temporär auf diese Daten zugreifen können, ohne deren Besitzer zu ändern. Dies verhindert Datenrennen (Data Races) und Speicherfehler.

5.2 Praktische Beispiele

5.2.1 Copy-Typen

Copy-Typen sind einfache Datentypen, die eine bitweise Kopie beim Zuweisen erstellen. Primitive Typen wie `i32` sind Beispiele dafür.

Listing 5.1: Copy-Typen

```
1 pub fn copy_type()
2 {
3     let i1 = 32;
4     let i2 = i1;
5
6     println!("{}", i1, i2);
7 }
```

Bei der Besitzübertragung wird der Besitz eines Datenobjekts von einer Variablen auf eine andere übertragen, was bedeutet, dass die ursprüngliche Variable nicht mehr auf das Objekt zugreifen kann.

Listing 5.2: Unveränderliche Ausleihe

```
1 pub fn transfer_ownership() {
2     let s1 = String::from("Hello");
3     // Ownership of the string is transferred from s1 to s2
4     let s2 = s1;
5
6     // This line causes a compile-time error because
7     // s1 no longer owns the string.
8     println!("{}", s1);
9     // This works perfectly, s2 now owns the data.
10    println!("{}", s2);
11 }
```

5.2.2 Ausleihe

Ausleihe ermöglicht es, dass eine Variable auf die Daten einer anderen Variable zugreift, ohne deren Besitzer zu ändern. Dies kann entweder als unveränderliche oder veränderliche Referenz erfolgen.

Listing 5.3: Unveränderliche Ausleihe

```
1 pub fn borrowing() {
2     let s1 = String::from("Hello");
3     // s2 is a reference to s1, s1 is borrowed
4     let s2 = &s1;
5
6     // s1 is still valid and hasn't been moved.
7     println!("{}", s1);
8     // s2 is a valid reference to s1.
9     println!("{}", s2);
10 }
```

5.2.3 Veränderliche Referenzen

Veränderliche Referenzen ermöglichen das Ändern der Daten, auf die sie zeigen, jedoch kann zu einem Zeitpunkt nur eine veränderliche Referenz existieren.

Listing 5.4: Veränderliche Referenz

```
1 pub fn mut_reference() {
2     let mut s1 = String::from("Hello");
3     // s2 is a mutable reference to s1
4     let mut s2 = &mut s1;
5
6     // only 1 mutable reference allowed
7     // will panic
8     let s3 = &mut s1;
9
10    // Modifying s1 through its mutable reference s2
11    s2.push_str(", world!");
12    // This works and prints "Hello, world!"
13    println!("{}", s2);
14 }
```

5.2.4 Kopieren und Klonen

Während primitive Typen kopiert werden, müssen komplexe Typen wie Strukturen explizit geklont werden, um eine vollständige Kopie zu erstellen.

Dies liegt daran, dass primitive Typen wie Ganzzahlen oder boolesche Werte eine feste, geringe Größe haben und direkt auf dem Stack gespeichert werden, wodurch das Kopieren effizient und unkompliziert ist. Komplexe Typen hingegen, wie Strukturen oder Vektoren, können größere und dynamische Daten enthalten, die auf dem Heap gespeichert werden. Das einfache Kopieren eines solchen komplexen Typs würde nur eine flache Kopie erstellen, die Referenzen auf dieselben Speicheradressen enthält. Um diese Problematik zu vermeiden und eine echte, tiefe Kopie zu erzeugen, muss die `Clone`-Trait implementiert und die `clone`-Methode aufgerufen werden [9] [10].

Listing 5.5: Kopieren und Klonen von Strukturen

```

1  #[derive(Debug, Clone)]
2  struct Book {
3      title: String,
4      pages: u32,
5  }
6
7  pub fn copy_move_clone_book() {
8      let num1 = 42; // i32, which is Copy
9      let num2 = num1; // num1 is copied to num2
10
11     // Both can be used; num1 was copied
12     println!("num1: {}, num2: {}", num1, num2);
13
14     let book1 = Book {
15         title: "Rust Programming".to_string(),
16         pages: 256,
17     };
18     // book1 is moved to book2
19     let book2 = book1;
20
21     // will cause a compile error because book1 has been moved
22     println!("book1: {:?}", book1);
23     // Only book2 can be used; book1 was moved
24     println!("book2: {:?}", book2);
25
26     let book3 = book2.clone();
27     println!("book2: {:?} book3: {:?}", book2, book3);
28 }

```

5.3 Vergleiche zu anderen Programmiersprachen

Im Gegensatz zu Rust erlauben viele andere Programmiersprachen wie C und C++ eine flexiblere, aber auch unsicherere Speicherverwaltung. In diesen Sprachen kann leicht unbeabsichtigter Code entstehen, der zu Speicherlecks oder Datenrennen führt da hier speicherplatz manuell allokiert werden muss und sich manuell um dessen freigabe gekümmert werden muss. Rusts strikte Regeln für Besitz und Ausleihe verhindern solche Probleme, indem sie

sicherstellen, dass es immer klar ist, wer für den Speicher eines Datenobjekts verantwortlich ist und wie darauf zugegriffen werden kann.

5.4 Sicherheitsvorteile der Besitz- und Ausleihregeln

Die strengen Regeln für Besitz und Ausleihe in Rust bieten mehrere Sicherheitsvorteile:

- **Vermeidung von Datenrennen:** Da immer nur eine veränderliche Referenz zu einem Zeitpunkt erlaubt ist, sind Datenrennen ausgeschlossen.
- **Speichersicherheit:** Rust verhindert durch seine Besitzregeln, dass Daten mehrfach freigegeben oder auf ungültigen Speicher zugegriffen wird.
- **Klarheit und Sicherheit:** Der Compiler erzwingt klare Regeln für den Zugriff auf Daten.

6 Lebensdauer-Tracking

6.1 Theoretische Grundlagen

Das Lebensdauer-Tracking ist ein zentrales Konzept in Rust, das dazu beiträgt, Speicherfehler zu verhindern. Lebensdauern (*lifetimes*) sind eine Art von generischen Parametern, die sicherstellen, dass Referenzen nur so lange gültig sind, wie es der Kontext erfordert. Dies verhindert das Auftreten von hängenden Referenzen und ermöglicht eine präzise Kontrolle über die Lebensdauer von Daten im Speicher.

Rust verwendet Lebensdauern, um sicherzustellen, dass Referenzen nicht auf ungültige Daten zeigen. Eine Lebensdauer beschreibt den Gültigkeitsbereich einer Referenz. Rust prüft zur Kompilierzeit, dass alle Referenzen gültig sind.

6.2 Praktische Beispiele

Hier sind zwei Beispiele, die zeigen, wie Lebensdauern in Rust verwendet werden:

Listing 6.1: Implizierte lifetime

```
1 pub fn get_first_word(s: &str) -> &str {  
2     s.split_whitespace().next().unwrap_or("")  
3 }
```

In diesem Beispiel wird eine Funktion definiert, die das erste Wort in einem `String` zurückgibt. Der Rückgabewert ist eine Referenz auf ein Teilstück des Eingabestrings. Rust stellt sicher, dass die Rückgabe nur so lange gültig ist, wie der Eingabestring selbst. In diesem Beispiel kann Rust die Lebensdauer aus dem Kontext heraus selbst bestimmen.

Listing 6.2: Explizierte lifetime

```
1 pub fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
2     if x.len() > y.len() {  
3         x  
4     } else {  
5         y  
6     }  
7 }
```

Hier wird eine Funktion definiert, die die längere von zwei Zeichenketten zurückgibt. Beide Eingabereferenzen und die Rückgabereferenz haben dieselbe Lebensdauer `'a`. Das bedeutet, dass die Rückgabereferenz nur so lange gültig ist, wie beide Eingabereferenzen gültig sind.

Die Angabe der Lebensdauer ist notwendig, da der Compiler ohne diese Annotation nicht in der Lage ist, die Beziehung zwischen den Lebensdauern der Eingabereferenzen und der Rückgabereferenz zu erkennen. Rusts *borrow checker* kann ohne die explizite Angabe der Lebensdauer `'a` nicht sicherstellen, dass die Rückgabe einer der Eingabereferenzen keine ungültigen Referenzen erzeugt. Die Lebensdauer `'a` stellt sicher, dass die Rückgabereferenz nur so lange gültig ist, wie beide Eingabereferenzen gültig sind, wodurch Speicherfehler vermieden werden.

6.3 Vergleich mit anderen Programmiersprachen

In vielen Programmiersprachen wie C und C++ muss der Programmierer manuell sicherstellen, dass Referenzen nicht auf ungültige Speicherbereiche zeigen, auch müssen diese manuell freigegeben werden.

Im Gegensatz dazu prüft Rust zur Kompilierzeit die Gültigkeit von Referenzen durch das Lebensdauer-Tracking. Dies führt zu sichererem Code, da viele Speicherfehler frühzeitig erkannt und verhindert werden.

6.3.1 Sicherheitsvorteile

Das Lebensdauer-Tracking in Rust bietet mehrere Sicherheitsvorteile:

- **Vermeidung von Dangling References:** Lebensdauern stellen sicher, dass Referenzen nur so lange gültig sind, wie die referenzierten Daten.
- **Kompilierzeitprüfungen:** Viele Speicherfehler werden bereits zur Kompilierzeit erkannt, was die Zuverlässigkeit des Codes erhöht.
- **Keine Garbage Collection:** Rust erreicht Speicher- und Referenzsicherheit ohne Garbage Collection, was zu besserer Performance führt.

Durch das Lebensdauer-Tracking stellt Rust sicher, dass Programme sicher und effizient arbeiten, ohne die typischen Probleme von Speicherverwaltung und Referenzen.

6.4 Fazit

Lebensdauer-Tracking ist ein mächtiges Werkzeug in Rust, das zur Speicher- und Referenzsicherheit beiträgt. Durch die strikte Kontrolle und Kompilierzeitprüfungen können viele gängige Fehler vermieden werden. Dies macht Rust zu einer Wahl für sicherheitskritische und performante Anwendungen.

7 Sichere Nebenläufigkeit

7.1 Theoretische Grundlagen

Nebenläufigkeit ist ein wesentlicher Aspekt moderner Softwareentwicklung, insbesondere bei der Entwicklung von Systemen, die eine hohe Leistung und Reaktionsfähigkeit erfordern. Jedoch birgt die Nebenläufigkeit zahlreiche Herausforderungen, insbesondere im Hinblick auf die Sicherheit und Korrektheit des Codes. Datenrennen sind ein häufiges Problem, das auftritt, wenn mehrere Threads gleichzeitig auf dieselben Daten zugreifen und diese ändern, ohne dass die Zugriffe korrekt synchronisiert sind.

Rust bietet einen Ansatz zur sicheren Nebenläufigkeit, indem es die Konzepte der Besitzrechte (Ownership) und des Ausleihens (Borrowing) auf Threads anwendet. Dies wird durch die Einbindung von Mechanismen wie **Arc** (Atomic Reference Counting) und **Mutex** (Mutual Exclusion) erreicht, die sicherstellen, dass Datenrennen verhindert werden, während gleichzeitig eine hohe Leistung erzielt wird.

Arc (Atomic Reference Counting) ist ein Thread-sicherer Referenzzähler, der es ermöglicht, dass mehrere Threads gleichzeitig Besitz an den gleichen Daten haben, indem er die Daten im Heap speichert und den Zähler atomar verwaltet. Dadurch bleibt der Speicher erhalten, solange mindestens eine Referenz darauf existiert.

Mutex (Mutual Exclusion) ist ein Synchronisationsprimitiv, das sicherstellt, dass nur ein Thread zu einem bestimmten Zeitpunkt auf die geschützten Daten zugreifen kann. Ein **Mutex** in Rust liefert einen **MutexGuard**, der den Zugriff auf die Daten kontrolliert und den **Mutex** freigibt, sobald der Guard außer Gültigkeit gerät.

Durch die Kombination von **Arc** und **Mutex** wird sichergestellt, dass veränderliche Daten sicher zwischen mehreren Threads geteilt werden können. **Arc** übernimmt das Zählen der Referenzen, während **Mutex** den exklusiven Zugriff auf die Daten garantiert. Dies funktioniert reibungslos mit Rusts Ownership- und Borrowing-System, da **Arc** die Besitzrechte an den Daten teilt und **Mutex** den Zugriff kontrolliert, um Datenrennen zu vermeiden.

7.2 Praktische Beispiele

Hier ist ein Beispiel, das zeigt, wie man in Rust sicher Threads erstellt, die gemeinsam einen Zähler erhöhen:

Listing 7.1: Sicheres Erstellen von Threads

```

1 use std::sync::{Arc, Mutex};
2 use std::thread;
3 use std::time::Duration;
4 use rand::{thread_rng, Rng};
5
6 pub fn create_increment_threads() {
7     let counter = Arc::new(Mutex::new(0));
8     let mut children = vec![];
9
10    for _ in 0..5 {
11        let counter_clone = Arc::clone(&counter);
12        let child = thread::spawn(move || {
13            let mut rng = thread_rng();
14            // Loop to add to the counter 10 times
15            for _ in 0..10 {
16                {
17                    let mut num = match counter_clone.lock() {
18                        Ok(x) => x,
19                        Err(_) => todo!(),
20                    };
21                    *num += 1;
22                } // MutexGuard goes out of scope here,
23                //releasing the lock
24
25                let sleep_time = rng.gen_range(1..=3);
26                // Sleep for random duration
27                thread::sleep(Duration::from_millis(sleep_time));
28            }
29        });
30        children.push(child);
31    }
32
33    // Wait for all threads to complete
34    for child in children {
35        child.join().unwrap();
36    }
37
38    // Print the result
39    println!("Final counter value: {}", *counter.lock().unwrap());
40 }

```

In diesem Beispiel wird ein gemeinsamer Zähler von mehreren Threads erhöht. Die Verwendung von `Arc<Mutex<i32>` stellt sicher, dass nur ein Thread gleichzeitig auf den Zähler zugreifen kann, wodurch Datenrennen verhindert werden.

Zum Vergleich zeigt das folgende Beispiel, wie man es *nicht* machen sollte. Es führt zu einem Kompilierfehler.

Listing 7.2: Unsicheres Erstellen von Threads

```

1 pub fn how_not_to_create_increment_threads() {
2     let mut counter = 0;
3     // need to use a reference
4     // otherwise int will be copied for each thread.
5     // Explicit mutable reference to 'counter'
6     let counter_ref = &mut counter;
7
8     let mut handles = vec![];
9
10    for _ in 0..10 {
11        // use the mutable reference in multiple threads
12        let handle = thread::spawn(move || {
13            // Error: 'counter_ref'
14            // cannot be sent safely between threads
15            *counter_ref += 1;
16        });
17        handles.push(handle);
18    }
19
20    for handle in handles {
21        handle.join().unwrap();
22    }
23
24    println!("Counter: {}", counter);
25 }
```

Dieses Beispiel zeigt einen unsicheren Ansatz, bei dem eine mutable Referenz zu einem Zähler in mehrere Threads übergeben wird. Dies führt zu einem Kompilierungsfehler, da Rusts Typensystem sicherstellt, dass solche unsicheren Zugriffe verhindert werden.

7.3 Vergleich mit anderen Programmiersprachen

Im Vergleich zu anderen Programmiersprachen wie C++ und Java bietet Rust erhebliche Sicherheitsvorteile. In C++ beispielsweise muss der Programmierer manuell sicherstellen, dass Zugriffe auf geteilte Daten korrekt synchronisiert werden, was fehleranfällig und schwierig ist. Java bietet eingebaute Synchronisationsmechanismen, aber diese sind oft mit Leistungseinbußen verbunden und weniger explizit in ihrer Nutzung.

Rusts Ansatz kombiniert die Leistungsvorteile von C++ mit der Sicherheit und Einfachheit der Synchronisation in Java, bietet jedoch eine explizitere und sicherere Kontrolle über die Nebenläufigkeit durch sein Besitz- und Ausleihmodell.

7.4 Sicherheitsvorteile

Die Sicherheitsvorteile von Rusts Ansatz zur Nebenläufigkeit sind erheblich:

- **Verhinderung von Datenrennen:** Durch die Verwendung von `Arc` und `Mutex` stellt Rust sicher, dass Datenrennen schon zur Kompilierzeit ausgeschlossen werden.

8 Fazit

Rust hat sich als eine Programmiersprache erwiesen, die Sicherheit und Leistung gleichermaßen priorisiert. Durch seine einzigartigen Sprachmerkmale bietet Rust Lösungen für viele der häufigsten Sicherheitsprobleme, die in der Softwareentwicklung auftreten. Die wichtigsten Sicherheitsaspekte, die wir in diesem Bericht untersucht haben, sind:

- **Statische Typen:** Rusts strenge statische Typisierung stellt sicher, dass Typfehler frühzeitig im Entwicklungsprozess erkannt werden. Dies reduziert die Wahrscheinlichkeit von Laufzeitfehlern und erhöht die Zuverlässigkeit des Codes.
- **Integer-Überlauf:** Rust verhindert Integer-Überläufe durch explizite Überlaufprüfungen und bietet Entwicklern Werkzeuge, um sichere numerische Berechnungen durchzuführen.
- **Zero-Cost-Abstraktionen:** Diese ermöglichen es, hochabstrakten und dennoch performanten Code zu schreiben. Rusts Abstraktionen verursachen keine Laufzeitkosten, was zu effizientem und sicherem Code führt. Gleichzeitig können z.B. *off-by-one* Fehler vermieden werden.
- **Fehlerbehandlung:** Rusts Ansatz zur Fehlerbehandlung ohne Ausnahmen, durch die Verwendung von `Result` und `Option`, fördert robustes und vorhersehbares Fehlermanagement.
- **Besitz und Ausleihe:** Das Besitz- und Ausleihmodell von Rust garantiert Speicher- und Datensicherheit ohne die Notwendigkeit eines Garbage Collectors. Dies verhindert Speicherlecks und Datenrennen.
- **Lebensdauer-Tracking:** Durch die explizite Angabe von Lebensdauern verhindert Rust Dangling References und gewährleistet sichere Speicherzugriffe.
- **Sichere Nebenläufigkeit:** Rust ermöglicht es, nebenläufigen Code sicher und effizient zu schreiben. Durch die Vermeidung von Datenrennen wird die Zuverlässigkeit nebenläufiger Programme erhöht.

Zusammenfassend lässt sich sagen, dass Rust durch diese Sprachmerkmale eine sichere und zuverlässige Entwicklung von Software ermöglicht. Die Kombination aus statischer Typisierung, sicherer Speicherverwaltung und robustem Fehlerhandling macht Rust zu einer idealen Wahl für sicherheitskritische Anwendungen.

Mit Blick auf die Zukunft verspricht Rust, durch kontinuierliche Weiterentwicklung und Verbesserung, seine Position als eine der sicheren und leistungsfähigen Programmiersprachen weiter zu festigen.

Beispielverzeichnis

1.1 i32 in i64	1
1.2 Signiertheit	1
2.1 Integer-Überläufe	3
3.1 Iterator	5
3.2 Message Passing	6
3.3 Option Typ	6
3.4 Filter-Funktion	7
4.1 Unsichere Fehlerbehandlung	8
4.2 Sichere Fehlerbehandlung	9
5.1 Copy-Typen	10
5.2 Unveränderliche Ausleihe	10
5.3 Unveränderliche Ausleihe	11
5.4 Veränderliche Referenz	11
5.5 Kopieren und Klonen von Strukturen	12
6.1 Implizierte lifetime	14
6.2 Explizierte lifetime	14
7.1 Sicheres Erstellen von Threads	17
7.2 Unsicheres Erstellen von Threads	18

Literaturverzeichnis

- [1] “Rust by example.” [Online]. Available: <https://doc.rust-lang.org/rust-by-example/>
- [2] D. Drysdale, *Effective Rust*. Sebastopol, CA: O’Reilly Media, Inc., 2024, copyright © 2024 Galloglass Consulting Limited. All rights reserved. Printed in the United States of America.
- [3] G. Shuklin, “My first insight into rust type system,” <https://medium.com/journey-to-rust/my-first-insight-into-rust-type-system-601cdfd0b81f>, 2020, published in Journey to Rust, Medium, Sep. 27, 2020.
- [4] MITRE, “CWE-190: Integer Overflow or Wraparound,” <https://cwe.mitre.org/data/definitions/190.html>, n.d., accessed: 2024-06-05.
- [5] Rust Foundation and the Rust Project Developers, *The Rust Programming Language*, 2nd ed. Rust Foundation, 2023, copyright © 2023 by the Rust Foundation and the Rust Project Developers.
- [6] L. Tung, “Microsoft: Here’s why we love programming language rust and kicked off project verona,” <https://www.zdnet.com/article/microsoft-heres-why-we-love-programming-language-rust-and-kicked-off-project-verona/>, 2020, may 22, 2020, 5:56 a.m. PT, ZDNet.
- [7] R. Sequeira, “Learning rust: Understanding zero-cost abstraction with filter and map,” <https://ranveersequira.medium.com/learning-rust-understanding-zero-cost-abstraction-with-filter-and-map-e967d09fff79>, 2023, may 23, 2023, Medium.
- [8] J. Haberman, “The overhead of abstraction in c/c++ vs. python/ruby,” <https://blog.reverberate.org/2014/10/the-overhead-of-abstraction-in-cc-vs.html>, 2014, october 17, 2014.
- [9] Rust Project Developers, “Clone trait,” <https://doc.rust-lang.org/std/clone/trait.Clone.html>, n.d., accessed: 2024-06-05.
- [10] Y. Feltzman, “Rust: Stack and heap, moves and copies,” <https://medium.com/israeli-tech-radar/rust-stack-and-heap-moves-and-copies-104b3146b977>, 2023, published in Israeli Tech Radar, July 24, 2023.

Glossar

borrow checker Ein System in der Programmiersprache Rust, das zur Kompilierzeit überprüft, ob alle Speicherzugriffe sicher sind. Es stellt sicher, dass Daten nie gleichzeitig sowohl mutabel als auch immutabel geborgt werden, und dass keine doppelten mutablen Referenzen existieren, um Race Conditions und andere Speicherfehler zu verhindern. i, 14

Datenrennen Ein Datenrennen tritt auf, wenn zwei oder mehr Threads in einem Computerprogramm gleichzeitig auf dieselbe Speicherstelle zugreifen, und mindestens einer der Zugriffe schreibend erfolgt, ohne dass die Zugriffe synchronisiert werden. Dies kann zu unerwarteten und fehlerhaften Verhaltensweisen führen, da die Reihenfolge der Zugriffe nicht garantiert ist und das Ergebnis von der Timing-Synchronisation der Threads abhängt.. i, ii

exception Eine *exception* (dt. Ausnahme) ist ein Mechanismus in vielen Programmiersprachen, der verwendet wird, um auf ungewöhnliche oder fehlerhafte Zustände während der Programmausführung zu reagieren. Ausnahmen treten auf, wenn ein Programm auf ein Problem stößt, das es nicht mit normalem Kontrollfluss handhaben kann, wie z.B. Division durch Null oder der Versuch, auf eine Datei zuzugreifen, die nicht existiert. Beim Auftreten einer Ausnahme wird der normale Ablauf des Programms unterbrochen und es wird eine spezielle Fehlerbehandlungsroutine aufgerufen. . i, 9

off-by-one Ein Fehler in der Programmierung, bei dem eine Schleife oder ein Array um eine Einheit zu viel oder zu wenig iteriert oder indiziert wird, was oft zu unerwartetem Verhalten führt. Dies passiert häufig bei der Implementierung von Schleifen oder bei der Arbeit mit Indizes, wo die korrekte Anzahl der Iterationen oder die korrekten Grenzen leicht übersehen werden können. i, 5, 7, 19

panic In der Programmiersprache Rust und anderen Programmiersprachen bezeichnet *panic* eine Laufzeitausnahme, die auftritt, wenn das Programm auf einen schwerwiegenden Fehler stößt, von dem es sich nicht erholen kann. Ein panic führt in der Regel dazu, dass das Programm sofort abgebrochen wird und die Kontrolle an das Betriebssystem zurückgegeben wird. Dies unterscheidet sich von regulären Fehlern, die durch Rückgabewerte oder Ausnahmen behandelt werden können. . i, 9

Tupel Ein Tupel ist eine geordnete Liste von Elementen, die in der Informatik häufig verwendet wird, um eine feste Anzahl von Werten zu speichern, die zu einer Einheit zusammengefasst sind. Die Elemente eines Tupels können unterschiedliche Datentypen haben. Tupel sind unveränderlich, was bedeutet, dass ihre Inhalte nach der Erstellung nicht mehr geändert werden können.. i, 4

Wrap-around Der Begriff "Wrap-around" beschreibt ein Phänomen, bei dem ein numerischer Wert, der seinen maximalen oder minimalen zulässigen Wert erreicht, auf den gegenüberliegenden Extremwert zurückgesetzt wird. Dies tritt häufig in der Computerprogrammierung auf, insbesondere bei der Arbeit mit beschränkten Datentypen wie Ganzzahlen. Ein Wrap-around kann zu Fehlern führen, wenn es nicht ordnungsgemäß gehandhabt wird.. i, 4