Fakultät Informatik

Sicherheitsaspekte Rust

Bericht im Studiengang Informatik

vorgelegt von
Robin Rosner

Matrikelnummer 362 5303

Abstract

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

Zunächst wird die statische Typisierung von Rust erläutert, die durch strikte Typenkontrolle zur Vermeidung typischer Fehler beiträgt. Anschließend wird gezeigt, wie Rust Integer-Überläufe verhindert und damit eine häufige Quelle von Sicherheitslücken eliminiert. 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 innovative 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 Dangling References minimiert.

Schließlich wird Rusts Ansatz zur sicheren Nebenläufigkeit untersucht, der Datenrennen 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 Sicherheitsfeatures von Rust und zeigt anhand praktischer Beispiele, wie diese in der Praxis angewendet werden können.

Inhaltsverzeichnis

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 ausgeführt werden. Statische Typisierung trägt erheblich zur Sicherheit und Stabilität von Programmen bei, da Typfehler und Typinkompatibilitäten bereits beim Kompilieren entdeckt werden.

1.2 Praktische Beispiele

Listing 1.1: i32 in i64

```rust
pub fn i32_in_i64() {
    let smol: i32 = 128;

    // panic: i32 will not get converted to i64
    let big: i64 = smol;

    // explicit conversion from i32 to i64
    let big: i64 = smol.into();

    println!("i32 in i64: {}", big);
}
```

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 expliziet bewusstsein für eine Typenkonvertierung und deren Risiken zu schaffen.

Listing 1.2: Signiertheit

```rust
pub fn signdness() {
    let unsigned_val: u32 = 150;
    let signed_val: i32 = -100;

    // will panic
    let signed: i32 = unsigned_val;
    let unsigned: u32 = signed_val;

    // explicit conversion from unsigned to signed
    // will not overflow in this context
    let signed_from_unsigned: i32 = unsigned_val as i32;
    println!("Signed from Unsigned: {}", signed_from_unsigned);

    // explicit conversion from signed to unsigned
    // has a logial error in this context and overflows
    let unsigned_from_signed: u32 = signed_val as u32;
    println!("Unsigned from Signed: {}", unsigned_from_signed);
    }
```

In diesem Beispiel zeigt die Funktion `signdness()` 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 Programierer explicit 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. Sprachen wie C und C++ bieten zwar statische Typisierung, aber ohne die strengen Sicherheitsgarantien wie bindende Typenumwandlung und automatischen Prüfungen von Rust, was zu Fehlern wie Pufferüberläufen 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.
- **Explizite Konvertierungen:** Automatische und implizite Typkonvertierungen sind in Rust selten, was dazu beiträgt, unerwartete Fehler zu vermeiden.
- **Strikte Typüberprüfung:** Der Compiler erzwingt strenge Regeln für Typen und ihre Verwendung, was zu sichererem und stabilerem Code führt.

Diese Eigenschaften machen Rust besonders geeignet für die Entwicklung von Systemsoftware und anderen sicherheitskritischen Anwendungen, bei denen Zuverlässigkeit und Sicherheit oberste Priorität haben.

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.

Rust hingegen bietet verschiedene Mechanismen, um Integer-Überläufe sicher zu handhaben und das Verhalten explizit zu definieren. 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: Beispielcode zur Behandlung von Integer-Überläufen in Rust

```
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, was zu undefiniertem Verhalten führen kann. Dies kann schwerwiegende Sicherheitslücken verursachen, die von Angreifern ausgenutzt werden können. Rust überprüft auf überlaufe im Debug-modus. Im Release-modus ist diese überpürung aus performance gründen standardmäßg deaktiviert. Eine über- oder unterlauf ist in Rust aber 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)**: Begrenzte 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 die explizite Handhabung von Integer-Überläufen 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.

3 Zero-Cost-Abstraktionen

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

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 genauso effizient bleibt wie bei direkter Implementierung.

3.2 Praktische Beispiele

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

- Iteratoren
- Message Passing
- 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-Beispiel

```
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 Beispiel

```
pub fn message_passing(){
    let (tx, rx) = mpsc::channel();

    let sender_thread = thread::spawn(move || {
        let msg = "Hello from the sender thread";
        // Sends a message safely to the receiver
        tx.send(msg).unwrap();
        println!("Sent message: '{}'", msg);
    });

    let receiver_thread = thread::spawn(move || {
        // Receives message safely
        let received = rx.recv().unwrap();
        println!("Received message: '{}'", received);
    });

    sender_thread.join().unwrap();
    receiver_thread.join().unwrap();
}
```

Durch eine solche einfach Abstraktion können Datenrennen zwischen threads vermieden werden. Trotzdem können Threads effektiv komunizieren 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 Beispiel

```
pub fn option_type(){
    let dividend = 10;
    let divisor = 0;

    match find_divisor(dividend, divisor) {
        Some(result) => println!("Result of division: {}", result),
        None => println!("Cannot divide by zero!"),
    }
}

fn find_divisor(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        None // Safely handle division by zero
    } else {
        Some(dividend / divisor) // Safely return the result
    }
}
```

Ein *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 Beispiel

```
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 seperiert 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 erheblich zur Sicherheit des Codes bei, da sie:

- Entwicklern ermöglichen, sichere und idiomatische Konstrukte zu verwenden, ohne sich um Leistungseinbußen sorgen zu müssen.
- Fehler und Sicherheitslücken reduzieren, da die Abstraktionen sicher und effizient implementiert sind.
- Den Code lesbarer und wartbarer machen, da Entwickler höhere Abstraktionslevel nutzen können, ohne auf Effizienz verzichten zu müssen.

4 Fehlerbehandlung

Rust bietet einen robusten Ansatz zur Fehlerbehandlung, der ohne Ausnahmen auskommt und somit 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. `Option<T>` repräsentiert einen optionalen Wert, der entweder `Some(T)` oder `None` sein kann.

- `Result<T, E>`: Wird verwendet, um den Erfolg oder Misserfolg einer Operation anzuzeigen.
- `Option<T>`: Wird verwendet, wenn ein Wert optional sein kann.

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

```rust
use std::fs::File;
use std::io::Read;

pub fn get_file() {
    match read_file_content() {
        Ok(contents) => println!("File contents: {}", contents),
        Err(e) => println!("Error reading file: {}", e),
    }
}

fn read_file_content() -> Result<String, std::io::Error> {
    let mut file = File::open("example.txt")?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

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 Program 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. Dies kann jedoch zu unvorhersehbarem Verhalten führen, insbesondere wenn Ausnahmen nicht ordnungsgemäß abgefangen werden. Rusts Ansatz, Fehler explizit zu behandeln, führt zu einem deterministischeren Verhalten und fördert den bewussten Umgang mit Fehlern.

- **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, was zu sichererem und robusterem Code führt.

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.

Zusammenfassend lässt sich sagen, dass Rusts Fehlerbehandlung einen entscheidenden Beitrag zur Sicherheit und Zuverlässigkeit von Software leistet. Durch die explizite Behandlung von Fehlern und die Vermeidung von Ausnahmen können viele typische Fehlerquellen eliminiert werden.

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 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 Beispiel

```
pub fn copy_type()
{
    let i1 = 32;
    let i2 = i1;

    println!("{}, {}", i1, i2);
}
```

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 Beispiel

```
pub fn transfer_ownership() {
    let s1 = String::from("Hello");
    // Ownership of the string is transferred from s1 to s2
    let s2 = s1;

    // This line causes a compile-time error because
    // s1 no longer owns the string.
    println!("{}", s1);
    // This works perfectly, s2 now owns the data.
    println!("{}", s2);
}
```

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 Beispiel

```
pub fn borrowing() {
    let s1 = String::from("Hello");
    // s2 is a reference to s1, s1 is borrowed
    let s2 = &s1;

    // s1 is still valid and hasn't been moved.
    println!("{}", s1);
    // s2 is a valid reference to s1.
    println!("{}", s2);
}
```

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 Beispiel

```
pub fn mut_reference() {
    let mut s1 = String::from("Hello");
    // s2 is a mutable reference to s1
    let mut s2 = &mut s1;

    // only 1 mutable reference allowed
    // will panic
    let s3 = &mut s1;

    // Modifying s1 through its mutable reference s2
    s2.push_str(", world!");
    // This works and prints "Hello, world!"
    println!("{}", s2);
}
```

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.

Listing 5.5: Kopieren und Klonen von Strukturen Beispiel

```
#[derive(Debug, Clone)]
struct Book {
    title: String,
    pages: u32,
}

pub fn copy_move_clone_book() {
    let num1 = 42; // i32, which is Copy
    let num2 = num1; // num1 is copied to num2

    // Both can be used; num1 was copied
    println!("num1: {}, num2: {}", num1, num2);

    let book1 = Book {
        title: "Rust Programming".to_string(),
        pages: 256,
    };
    // book1 is moved to book2
    let book2 = book1;

    // will cause a compile error because book1 has been moved
    println!("book1: {:?}", book1);
    // Only book2 can be used; book1 was moved
    println!("book2: {:?}", book2);

    let book3 = book2.clone();
    println!("book2: {:?} book3: {:?}", book2, book3);
}
```

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. 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, was zu sichererem und leichter verständlichem Code führt.

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 Dangling References 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, wodurch viele gängige Speicherfehler verhindert werden.

6.2 Praktische Beispiele

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

Listing 6.1: Ermitteln des ersten Wortes

```
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: Längere von zwei Zeichenketten ermitteln

```
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. Dies führt oft zu Speicherfehlern wie Dangling References und Speicherlecks oder geringere Performance mit einem Garabe Collector.

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. Durch das explizite Anfordern und Freigeben von Sperren mit `MutexGuard` bleibt der Code dennoch performancestark und sicher.

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 mit Rust

```rust
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;
use rand::{thread_rng, Rng};

pub fn create_increment_threads(){
    let counter = Arc::new(Mutex::new(0));
    let mut children = vec![];

    for _ in 0..5 {
        let counter_clone = Arc::clone(&counter);
        let child = thread::spawn(move || {
            let mut rng = thread_rng();
            // Loop to add to the counter 10 times
            for _ in 0..10 {
                {
                    let mut num = match counter_clone.lock() {
                        Ok(x) => x,
                        Err(_) => todo!(),
                    };
                    *num += 1;
                } // MutexGuard goes out of scope here,
                  //releasing the lock

                let sleep_time = rng.gen_range(1..=3);
                // Sleep for random duration
                thread::sleep(Duration::from_millis(sleep_time));
            }
        });
        children.push(child);
    }

    // Wait for all threads to complete
    for child in children {
        child.join().unwrap();
    }

    // Print the result
    println!("Final counter value: {}", *counter.lock().unwrap());
}
```

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 kann, es gibt einen compile error:

Listing 7.2: Unsicheres Erstellen von Threads in Rust

```rust
pub fn how_not_to_create_increment_threads(){
    let mut counter = 0;
    // need to use a reference
    // otherwise int will be copied for each thread.
    // Explicit mutable reference to `counter`
    let counter_ref = &mut counter;

    let mut handles = vec![];

    for _ in 0..10 {
        // use the mutable reference in multiple threads
        let handle = thread::spawn(move || {
            // Error: `counter_ref`
            // cannot be sent safely between threads
            *counter_ref += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Counter: {}", counter);
}
```

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 Datenrassen auf kompilierbarem Weg verhindert werden.

- **Explizite Synchronisation:** Rust erfordert, dass alle Synchronisationspunkte explizit definiert werden, was die Lesbarkeit und Wartbarkeit des Codes erhöht.
- **Hohe Leistung:** Trotz der zusätzlichen Sicherheit bietet Rust eine hohe Leistung, da es unnötige Synchronisationskosten vermeidet.

8 Fazit

Rust hat sich als eine herausragende 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.
- **Fehlerbehandlung:** Rusts Ansatz zur Fehlerbehandlung ohne Ausnahmen, durch die Verwendung von `Result` und `Option`, fördert robustes und vorhersehbares Fehlermanagement.
- **Besitz und Ausleihe:** Das innovative 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. Die in diesem Bericht vorgestellten Sicherheitsaspekte zeigen, dass Rust nicht nur theoretische, sondern auch praktische Lösungen für viele der größten Herausforderungen der Softwareentwicklung bietet.

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

List of Listings

Literaturverzeichnis

[Rust] “Rust by Example”.