Rust in 4 Wochen

HSLU PCP Team-Projekt

Vision, Geschichte & Verbreitung

Rust wurde von Mozilla Research entwickelt, um eine Alternative zu C und C++ zu schaffen, die sicher, schnell und gleichzeitig modern ist. Die erste stabile Version erschien 2015. Seitdem hat sich Rust zu einer beliebten Sprache für System-programmierung, WebAssembly und eingebettete Systeme entwickelt. Besonders hervorzuheben ist die hohe Sicherheit bei gleichzeitiger Performance, das ganze ohne Garbage Collector. Die Sprache wird regelmässig zur «beliebtesten Programmiersprache» auf Stack Overflow gewählt, wobei hervorzuheben ist, dass trotz ihrer Beliebtheit, nicht viele Stellen für Rust-Entwickler auf dem Schweizer Markt zu finden sind.

Fokuspunkte

Panic!

Rust hat einen sehr strikten Compiler. Wie in den folgenden Kapiteln festgestellt werden kann, wird sehr viel nicht lauffähiger Code bereits zur Compile-Time auffallen. Leider können nicht alle Laufzeitfehler zur Compile-Time festgestellt werden, beispielsweise ein Zugriff auf einen Index eines Arrays, welcher über die Grenzen des Arrays hinausgeht. In Solchen Fällen verfällt Rust in eine «Panik» und stoppt das Programm. Dies kann auch manuell forciert werden mit der «panic!»-Macro.
Dabei werden stets alle Ressourcen des panischen Threads sicher freigegeben.

Borrowing & Move Semantik

Rust verwendet ein Ownership-System, um Speicher sicher und effizient zu verwalten, ganz ohne Garbage Collector. Jeder Wert hat genau einen Besitzer, und sobald dieser aus dem Scope fällt, wird der Speicher freigegeben.

Dieses System wird auch vom Borrow-Checker, eine Komponente des Compilers sichergestellt. Code, welcher das Ownership-System verletzt, kann dementsprechend schon gar nicht kompiliert werden.
Hier gibt es aber Umwege, welche wir nicht weiter behandeln werden. Diese können im Buch «The Rustonomicon – The Dark Arts of Unsafe Rust» nachgeschlagen werden.

Um das Verhalten dieses Ownership System zu verstehen, muss man verstehen, wie und vor Allem wo welche Art von Werten gespeichert werden. Primitive Werte wie Integers oder Booleans liegen auf dem Stack und werden als Call-By-Value in Funktionen übergeben, bei diesen Werten spielt Ownership keine Rolle, da der «Besitzer» eines Integers eine Kopie des Wertes an eine Funktion übergibt und beide Variablen unabhängig voneinander weiterleben.
Komplexere Werte wie String besitzen über Daten auf dem Heap. In Funktionen kann der Besitzer einer solchen Wertes entweder explizit übergeben werden (Call-By-Value,

Ownership wird transferiert) oder nur eine Referenz (&T) auf den Wert (Call-by-Reference) übergeben werden. Das Ownership-System stellt sicher, dass dabei keine doppelten Besitzer existieren.

Dies lässt sich in den folgenden Abschnitten an Beispielen von Strings, Ein Datentyp, welcher auf dem Heap liegt, darstellen.

Move-Verhalten

Ein Bestandteil dieses Ownership-Modells ist das Move-Verhalten: Übergibt man einen Wert ohne Referenz (&), wird die Ownership übertragen. Danach kann die ursprüngliche Variable nicht mehr verwendet werden.

Nachfolgend ein Beispiel für eine Ownership übernehmende Funktion.

```
fn print_heap_value_without_returning_ownership(string: String) {
    println!("The function now has ownership over the string, it is moved into here: {string}");
}
```

Borrowing

Mit Borrowing kann man Referenzen übergeben, ohne Ownership zu übertragen. Dabei wird zwischen mutable und immutable Referenzen explizit unterschieden.
Nachfolgend 2 Beispiele für Funktionen mit mutable und immutable Referenzen.

```
fn print_heap_value_by_borrowing(string: &String) {
    println!("The function borrowed a readonly reference of the string: {string}");
}
fn mutable_borrow(string: &mut String) {
    *string = String::from( s: "I have been mutated");
}
```

Rust erzwingt dabei: Es darf zu jedem Zeitpunkt entweder mehrere immutable Referenzen oder genau eine mutable Referenz geben, aber nie beides gleichzeitig.

Typestate Programming

Beim Typestate-Pattern wird der Zustand eines Objekts im Typen-System codiert. So stellt der Compiler sicher, dass Methoden nur im richtigen Zustand aufgerufen werden können.

In den angehängten Code Beispielen kann dieses Pattern betrachtet werden. Eine Verbindung kann nur aufgebaut werden, wenn vorher connect() aufgerufen wurde. Erst danach erlaubt der Compiler den Aufruf von build(), und nur nach erfolgreichem Aufbau ist send() verfügbar.

```
fn main() {
    // Start disconnected
    let conn = ConnectionBuilder::new()
        .connect()  // only after this, you can build
        .build();

    // Now it's safe to use
    conn.send("Hello typestate!");
}
```

Dies erhöht die Sicherheit zur Compile-Zeit und verhindert Zustandsfehler zur Laufzeit.

Patterns & Matching

Rusts Pattern Matching ermöglicht es, komplexe Datenstrukturen wie Enums oder Structs präzise zu analysieren und zu verarbeiten. Vergleichbar mit match in funktionalen Sprachen.

```
match msg {
    Message::Quit => { ... }
    Message::Move { x :i32 , y :i32 } => { ... }
    Message::Write(text :String ) => { ... }
    Message::ChangeColor(r :u8 , g :u8 , b :u8 ) => { ... }
}
```

Alternativ erlaubt if let oder let else eine schlanke Syntax für Einzelfälle:

```
let Some(x :u8 ) = number else {
    println!("Kein Wert vorhanden");
    return;
};
```

Pattern Matching ist nicht nur leserlich, sondern auch sicher: Der Compiler prüft ob alle Fälle abgedeckt sind.

Spawns & Channels

Bei diesem Feature besteht die Möglichkeit, Nebenläufigkeit sicher und einfach umzusetzen. Dabei kommen zwei wichtige Konzepte zum Einsatz: *spawn* zum Starten von Threads und *channel* zur Kommunikation zwischen ihnen.

Mit std::thread::spawn kann ein neuer Thread gestartet werden. Um Daten sicher zwischen Threads zu übertragen, nutzt man std::sync::mpsc::channel. Dabei steht mpsc für Multiple Producer, Single Consumer. Es können also mehrere Sender, aber nur ein Empfänger existieren.

```
// Erstelle einen Kanal (Sender, Empfänger)
let (tx, rx) = mpsc::channel();

// Starte einen neuen Thread und sende eine Nachricht
thread::spawn(move || {
    let message = "Hallo von einem anderen Thread!";
    println!("Child-Thread: sende Nachricht…");
    tx.send(message).unwrap(); // sendet über den Kanal
    thread::sleep(Duration::from_secs(1));
    println!("Child-Thread: fertig.");
});

// Haupt-Thread wartet auf Nachricht
println!("Haupt-Thread: warte auf Nachricht…");
let received = rx.recv().unwrap();
println!("Haupt-Thread: erhalten -> {}", received);
```

In diesem Beispiel wartet der Haupt-Thread blockierend mit recv(), während der Child-Thread eine Nachricht mit send() überträgt. Durch das move-Schlüsselwort wird der

Sender (tx) in den neuen Thread verschoben. Das Ownership-System von Rust verhindert dabei ein Data Race bereits zur Compile-Zeit. Das heisst, es wird verhindert, dass zwei oder mehr Threads nicht gleichzeitig auf denselben Speicherbereich zugreifen.

Fazit

Team-Fazit

Als Team haben wir Rust als moderne und durchdachte Systemsprache kennengelernt. Besonders beeindruckt hat uns die strikte Compile-Time-Sicherheit durch das Ownership- und Borrowing-Modell. Viele typische Fehlerquellen werden bereits beim Kompilieren ausgeschlossen, was zu robustem und sicherem Code führt.

Trotz anfänglicher Lernkurve ermöglicht Rust dank einer starken Standardbibliothek und klarer Sprachkonzepte eine effiziente Entwicklung. Gerade für Anwendungen mit Anforderungen an Performance, Parallelität und Sicherheit sehen wir in Rust ein grosses Potenzial. Sowohl im System als auch im Anwendungsbereich.

Fazit Kevin Wüstner

Rust war für mich eine spannende Erfahrung, besonders im Vergleich zu abstrakteren Sprachen wie Prolog. Ich schätze die vollständige Kontrolle ohne «Black Magic», die strikte Trennung zwischen Statements und Expressions sowie die standardmässige Immutability. Diese Aspekte fördern einen funktionalen Programmierstil, den ich bevorzuge, weil er Änderungen im Code nachvollziehbarer macht. Der Compiler ist zwar streng, aber hilfreich, und die integrierte Dokumentation in Form des Buchs zum Lernen der Sprache macht den Einstieg trotz hoher Komplexität gut machbar. Rusts Praxisrelevanz scheint mir jedoch (zumindest im Schweizer Markt) noch begrenzt, obwohl man sehr viel in Artikeln und Blogposts über verschiedenste Rust-Rewrites hört.

Fazit Livio Theiler

Da ich bereits Erfahrung mit C++ habe, war Rust für mich besonders spannend im direkten Vergleich. Viele Konzepte wie Ownership und manuelle Speicherverwaltung sind mir vertraut, doch Rust hebt diese auf ein neues, sichereres Niveau. Besonders beeindruckt hat mich der Compiler, der viele potenzielle Fehler schon zur Compile-Zeit erkennt und gleichzeitig hilfreiche Hinweise liefert.

Insgesamt war die Arbeit mit Rust eine wertvolle technische Erfahrung. Auch wenn Rust im aktuellen beruflichen Umfeld wohl eher selten zum Einsatz kommt, sehe ich das Gelernte als wertvolles Fundament, insbesondere in Bereichen, in denen Sicherheit und Performance entscheidend sind.