Rust

Programming Concepts and Paradigms (HS 2018)

Lukas Arnold & Patrick Bucher

07.12.2018

Inhaltsverzeichnis

1 Installation

Eine Rust-Installation beinhaltet folgende Werkzeuge:

- `rustc`: Compiler
- `cargo`: Package- und Build-Management
- `rustdoc`: Erstellung von Dokumentation

1.1 Voraussetzungen

Unter macOS und Linux wird ein Linker benötigt und ein C-Compiler benötigt. `gcc` und `llvm` sind weit verbreitete Optionen.

Für Windows wird das Microsoft Visual C++ Redistributable for Visual Studio 2017 benötigt:

- Version für 64-Bit-Architektur
- Version für 32-Bit-Architektur

1.2 macOS and Linux: Mittels `rustup`

Die einfachste Variante ist die Installation mit `rustup`, wozu man folgende Befehlszeile mit der Shell ausführen muss:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Anschliessend kann man den Instruktionen folgen. Die `curl`-Parameters sind nötig, um zu verhindern dass `sh` Zeichen ausgaben erhält, womit es nichts anfangen kann:

- `-s`: silent (keine Statusmeldungen ausgeben)
- `-S`: show errors (nur Fehler anzeigen)
- `-f`: fail silently (bei serverseitigen Fehlern keine Ausgabe produzieren)

Damit die Umgebungsvariablen nach der Installation aktualisiert werden kann man entweder eine neue Shell öffnen oder folgende Befehlszeile ausführen:

```
$ source $HOME/.cargo/env
```

Anschliessend kann man die Installation folgendermassen überprüfen:

```
$ rustc --version
rustc 1.30.1 (1433507eb 2018-11-07)
$ cargo version
cargo 1.30.0 (a1a4ad372 2018-11-02)
$ rustdoc --version
rustdoc 1.30.1 (1433507eb 2018-11-07)
```

Aktualisierungen und können mit `rustup` durchgeführt werden:

```
$ rustup update
```

Rust und `rustup` könne folgendermassen wieder deinstalliert werden:

```
$ rustup self uninstall
```

1.3 macOS and Linux: Mittels Package Manager

macOS mit Homebrew und viele Linux-Distributionen bieten eigene Packages für Rust an. Nach der Installation sollte man wie gerade beschrieben sicherstellen, dass zumindest die Programme `rustc`, `cargo` und `rustdoc` installiert wurden.

Homebrew (macOS):

```
$ brew install rust
```

aptitude (Debian, Ubuntu):

```
$ aptitude install rustc cargo rust-doc
```

pacman (Arch Linux):

```
$ pacman -S rust rust-docs
```

1.4 Windows: Mittels `rustup-init`

Für Windows lässt sich Rust über ein Setup-Programm installieren:

1. `rustup-init.exe` von win.rustup.rs herunterladen.
2. Das Programm ausführen und den Anweisungen folgen.
3. Eine längere Pause einplanen, da die Installation der Dokumentation auf Windows bekanntermassen wesentlich länger dauert als auf macOS and Linux.
4. Die Installation mittels `cmd.exe` oder Powershell validieren (siehe oben).

2 Geschichte

Die Geschichte von Rust lässt sich grob in fünf Phasen einteilen:

1. 2006-2010: Rust begann als privates Projekt des Mozilla-Mitarbeiters Graydon Hoare. Der Compiler war anfangs in OCaml geschrieben. Einige wichtige Merkmale der Sprache gehen auf diese Phase zurück: kurze Schlüsselwörter, Type Inference, Generics, Memory Safety, keine `null`-Werte. Die Sprache sollte mehrere Programmierparadigmen unterstützen, aber nicht besonders objektorientiert sein. Damals verfügte Rust noch über einen Garbage Collector.
2. 2010-2012: Mozilla nahm Rust unter seine Obhut und liess ein kleines Team um Graydon Hoare daran arbeiten. Der Firefox-Browser bestand damals aus ca. 4.5 Millionen Zeilen C++-Code, und Änderungen am Code führten oft zu Fehlern. Rust wurde nun mit dem Ziel weiterentwickelt, dass der Compiler menschliche Fehler möglichst verhindern soll, zumindest was Memory-Management und Race-Conditions bei nebenläufiger Programmierung betrifft. Zu dieser Zeit wurde das Typsystem stark weiterentwickelt. Viele Features vom Sprachkern wurden in Libraries verschoben. Der Garbage Collector wurde nicht mehr benötigt.
3. 2012-2014: Das Paketverwaltungs- und Buildwerkzeug `cargo` und die Plattform Crates.io für die Publikation von Libraries wurden erstellt. Graydon Hoare verlässt Mozilla und zieht sich aus der Weiterentwicklung von Rust zurück. Die Community spielte zusehends eine wichtigere Rolle, und ein RFC-Prozess (Request for Comments) für die Weiterentwicklung von Rust wurde initiiert. Zustrom erhielt die Community aus verschiedenen Lagern:
   - C++: Programmierer, die hardwarenah programmieren wollen und an einer hohen Performance interessiert sind.
   - Skriptsprachen: Programmierer, die an zeitgemässem Tooling und an einer bequemen Entwicklungsprozess interessiert sind.
   - Funktionale Programmiersprachen: Programmierer, die sich für das Typsystem und funktionale Features interessieren.
4. 2015-2016: Bei Rust gab es in den frühen Jahren oft viele nicht rückwärtskompatible Änderungen. Seit Version 1.0.0, die am 15. Mai 2015 veröffentlicht wurde, sodass die

Community mit einer weitgehendst stabilen Sprache auf die Weiterentwicklung der Libraries konzentrieren kann. Der Release-Plan sieht kleinere Veröffentlichungen alle sechs Wochen vor. Mit `rustfmt` soll Rust einen einheitlichen Code-Formatter erhalten, wie es Go mit `gofmt` sehr erfolgreich vormachte.

5. seit 2016: Mit einem neu entwickelten mp4-Parser erhält Rust erstmals Einzug in den Firefox-Browser. Die Rendering-Engine für Firefox wird neu in Rust geschrieben (Servo). Dropbox verwendet ein Dateisystem, das in Rust geschrieben ist. Mit Redox wird ein Unix-artiges, experimentelles Betriebssystem in Rust entwickelt.

3 Pattern Matching

Viele Programmiersprachen verwenden `null`, um die Abwesenheit eines Wertes zu signalisieren. Das Konzept ist an sich sinnvoll, nur erlauben es die meisten Programmiersprachen, dass `null` (oder `nil`, oder `None`) verwendet werden kann, als ob es ein normaler Wert wäre. Dies führt zu schwerwiegenden Laufzeitfehlern.

Rust kennt kein `null`. Stattdessen wird die An- und Abwesenheit eines Wertes mit der Enumeration `Option<T>` gelöst, welche folgendermassen definiert ist:

```
enum Option<T> {
    Some(T),
    None,
}
```

Die Enumeration `Option` hat einen Typparameter `T`, ist also generisch. Die *Variante* `Some` hält einen Wert des entsprechenden Typs `T`. Die Variante `None` hat keinen Wert. Auf den ersten Blick sieht es so aus, dass das Problem mit `null` nur verschoben und mit `None` anders bezeichnet wird. Der Rust-Compiler stellt jedoch sicher, dass ein Ausdruck vom Typ `Option<T>` nicht einfach so wie ein Ausdruck vom Typ `T` verwendet werden kann. Betrachten wir folgende Funktion, die zwei Ganzzahlen dividiert und das Ergebnis als `Option` einer Gleitkommazahl zurückgibt:

```
fn divide(a: i32, b: i32) -> Option<f64> {
    if b == 0 {
        return None;
    }
    return Some(a as f64 / b as f64);
}
```

Wird die Rückgabe wie ein normaler Wert behandelt, kann das Programm nicht kompiliert werden:

```
let x = 10 + divide(10, 3);
```

```
$ cargo check
let x = 10 + divide(10, 3);
```

```
  ^ no implementation for `{integer} + std::option::Option<f64>`
```

Der Ausdruck vom Typ `Option<f64>` muss zuerst entpackt werden, um an den Wert vom Typ `f64` heranzukommen. Hier kommt das *Pattern Matching* ins Spiel, womit ein `enum`-Ausdruck der jeweiligen Variante zugeordnet wird:

```
match divide(10, 3) {
    Some(x) => println!("{}", x),
    None => println!("fail"),
}
```

`match` ist vergleichbar mit `switch` in Java und C. Im Kontext mit `enum`-Ausdrücken stellt der Rust-Compiler jedoch sicher, dass jede Variante in einem eigenen *Arm* behandelt wird. Würde im obigen Beispiel der zweite Arm weggelassen, scheiterte die Kompilierung:

```
match divide(10, 3) {
      ^^^^^^^^^^^^ pattern `None` not covered
```

Mittels Pattern Matching eliminiert Rust eine weitere Fehlerklasse: das unvollständige Abdecken von Fällen.

Ein weiteres Anwendungsbeispiel von `match` ist der Vergleich von Zahlen bzw. deren Sortierung. In Java gibt ein Ausdruck `a.compareTo(b)` bei `a>b` eine positive Zahl, bei `a<b` eine negative Zahl und bei `a==b` die Zahl null zurück. Bei Rust retournieren Vergleiche eine `enum` vom Typ `Ordering`. Im folgenden Beispiel wird eine geratene Zahl `guess` mit einer zuvor festgelegten Zahl `secret_number` verglichen:

```
match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal=> println!("Right guess!"),
}
```

So wird wiederum sichergestellt, dass auf alle möglichen Ergebnisse reagiert wird. Das Weglassen eines Armes würde wiederum zu einem Kompilierfehler führen.