Rust

Programming Concepts and Paradigms (HS 2018)

Lukas Arnold & Patrick Bucher

13.12.2018

Inhaltsverzeichnis

1 Installation

Eine Rust-Installation beinhaltet folgende Werkzeuge:

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

1.1 Voraussetzungen

Unter macOS und Linux wird ein Linker 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` Ausgaben erhält, mit denen 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 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 worden sind.

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`

Auf 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).

1.5 Entwicklungsumgebung

Als Entwicklungsumgebung haben wir `vim` (ohne weitere Plugins) und Visual Studio Code mit der offiziellen Rust-Erweiterung (basierend auf dem Rust Language Server) verwendet. Für Code-Vervollständigung steht das IDE-unabhängige Werkzeug Rust Racer zur Verfügung, das bei `vim` mit dem Plugin Vim Racer eingebunden werden könnte. IntelliJ-Benutzer können die Erweiterung IntelliJ Rust verwenden. Aktuelle Informationen über die IDE-Unterstützung von Rust erhält man auf AreWeIDEyet.

2 Geschichte

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

1. 2006-2010: Rust beginnt als privates Projekt des Mozilla-Mitarbeiters Graydon Hoare. Der Compiler ist 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 soll mehrere Programmierparadigmen unterstützen, aber nicht besonders objektorientiert sein. Rust verfügt noch über einen Garbage Collector.
2. 2010-2012: Mozilla nimmt Rust unter seine Obhut und lässt ein kleines Team um Graydon Hoare daran arbeiten. Der Firefox-Browser besteht aus ca. 4.5 Millionen Zeilen C++-Code, und Änderungen am Code führen oft zu Fehlern. Rust wird mit dem Ziel weiterentwickelt, dass der Compiler menschliche Fehler möglichst verhindern soll, gerade was Memory-Management und Race-Conditions bei nebenläufiger Programmierung betrifft. Das Typsystem wird stark weiterentwickelt. Viele Features vom Sprachkern werden in Libraries ausgelagert. Der Garbage Collector wird nicht mehr benötigt.
3. 2012-2014: Das Paketverwaltungs- und Buildwerkzeug `cargo` und die Plattform Crates.io für die Publikation von Libraries entstehen. Graydon Hoare verlässt Mozilla und zieht sich aus der Weiterentwicklung von Rust zurück. Die Community spielt zusehends eine wichtigere Rolle, und ein RFC-Prozess (Request for Comments) für die Weiterentwicklung von Rust wird initiiert. Zustrom erhält die Community aus verschiedenen Lagern mit unterschiedlichen Zielen:
    - C++: Programmierer, die hardwarenah programmieren wollen und an einer hohen Performance interessiert sind.
    - Skriptsprachen: Programmierer, die sich zeitgemässes Tooling und einen bequemen Entwicklungsprozess wünschen.
    - Funktionale Programmiersprachen: Programmierer, denen ein gutes Typsystem und funktionale Features wichtig sind.
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 wird, kann sich die Community mit einer weitgehendst stabilen Sprache auf die Weiterentwicklung der Libraries konzentrieren. 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 hielt Rust erstmals Einzug in den Firefox-Browser. Die neue CSS-Engine für Firefox Stylo ist in Rust geschrieben. Sie ist ein Teil der Rendering-Engine Servo, die ebenfalls in Rust entwickelt wird. Dropbox verwendet Rust für das Dateisystem. Mit Redox wird ein experimentelles, Unix-artiges Betriebssystem in Rust entwickelt.

3 Ownership

Das Ownership-Konzept in Rust ist ein wenig speziell und schwer mit anderen Sprachen zu vergeleichen. Das Ziel des Konzeptes ist es ganz klar zu definieren, wer für eine Variable zuständig und wie lange eine Variable gültig ist. Durch des strenge Ownership-Konzept,

welches durch den Compiler erzwungen wird, sollte es keine Problem mit Pointern geben.

Das Ownership-Konzept von Rust wird durch die folgenden 3 Regeln definiert:

1. Jeder Wert in Rust gehört zu einer Variablen, welche ihr Owner genannt wird.
2. Es kann zur gleichen Zeit immer nur einen Owner geben.
3. Wenn der Owner den Scope verlässt, wird der Wert gelöscht.

3.1 Heap vs. Stack

Um das Ownership-Konzept zu verstehen muss jedoch zuerst der Unterschied zwischen Stack und Heap klar sein. Der Stack ist ein streng organisierter Speicherbereich mit einem schnellen Zugriff. Die Daten werden gemäss LIFO-Prinzip (last in, first out) abgelegt und es können nur Daten mit einer fixen Grösse abgelegt werden.

Der Heap dagegen kann gebraucht werden für Daten, bei denen die Grösse zum Kompilierungszeitpunkt noch nicht bekannt ist. Daher ist der Heap weniger stark organisiert, da das Betriebsystem zur Laufzeit frei Speicherbereiche finden muss. Dies führt natürlich zu längeren Zugriffszeiten.

3.2 Skalare Datentypen

Bei skaleren Datentypen verhält sich Rust so, wie man es sich von anderen Sprachen gewohnt ist. Zum Beispiel gibt der folgende Code den Wert 5 auf der Konsole aus. Das liegt daran, dass diese Datentypen bei einer Zuweisung oder einem Funktionsaufruf kopiert werden.

```
let x = 5;
let y = x;
println!("{}", y);
```

Die folgenden Typen implementieren das Trait `Copy` und werden daher automatisch bei einem Aufruf oder einer Zuweisung kopiert:

- alle Integer-Datentypen, wie z.B. `u32`
- alle Fliesskommazahlen, wie z.B. `f64`
- Booleans (`bool`) und Zeichen (`char`)
- Tuples, bei denen all Datentypen das `Copy`-Trait implementieren

3.3 Komplexere Datentypen

Bei einem sehr ähnlichen Beispiel wie dem vorherigen verhält sich der Code jedoch nicht so, wie man es erwarten könnte, denn der Code lässt sich nicht kompilieren. Der Compiler gibt dabei folgende Fehlermeldung zurück, welche genau auf das oben angesprochene Problem hinweist. Der Owner eines `String` wird bei einer Zuweisung oder einem Funktionsaufruf verändert, und daher ist der alte Variablenname nicht mehr gültigt.

```rust
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

```
error[E0382]: use of moved value: `s1`
--> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  | value used here after move ^^
  |
  = note: move occurs because `s1` has type `std::string::String`,
          which does not implement the `Copy` trait
```

Der Grund für dieses Verhalten ist, dass Strings zum Komplierungszeitpunkt keine fixe Grösse haben, da diese während dem Verlauf des Programms geändert werden können. Daher werden Strings im Heap gespeichert und nicht automatisch bei einer Verwendung kopiert.

Damit man dennoch beispielsweise einen `String` einer Methode übergeben kann, stellt Rust ein Konzept namens Borrowing zur Verfügung. Dabei wird einer Funktion nicht der Wert einer Variablen übergeben sondern eine Referenz. Auf dieser Referenz können dann Operationen ausgeführt werden, aber der Owner der Daten bleibt weiterhin die vorherige Variable.

```rust
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
    println!("{}", s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", from 06");
}
```

4 Traits

Das Konzept von Traits ist sehr ähnlich zu Interfaces in anderen Sprachen. Es gibt jedoch einige Unterschiede. Ein Trait erlaubt es, gleiches Verhalten für einen bestimmten Zweck zu gruppieren. Ein Trait gibt die Funktionsignaturen vor, welche für die Nutzung implementiert werden sollen. Es besteht auch die Möglichkeit, dass ein Trait für Methoden eine Standard-Implementierung zur Verfügung stellen kann.

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Tweet {
    pub username: String,
    // ...
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}", self.username)
    }
}
```

Traits entfalten ihr volles Potential erst wenn man sie benutzt um bestehenden Klasse neues Verhalten anzufügen. Das heisst in Rust gibt es die Möglichkeit bestehende Klassen (eigene oder aus der Standardbibliothek) für die Verwendung im eigenen Code zu erweitern. Im folgenden Beispiel wird beispielsweise der Klasse `String` aus der Standardbibliothek eine Methode hinzugefügt, welche das erste Wort des Strings zurückgibt. Die Implementation ist sehr minimalistisch und sollte hauptsächlich die Möglichkeit veranschaulichen.

```rust
pub trait FirstWord {
    fn first_word(&self) -> String;
}

impl FirstWord for String {
    fn first_word(&self) -> String {
        let parts: Vec<_> = self.split(" ").collect();
        parts[0].to_string()
    }
}

fn main() {
    let s = String::from("hello world");
    println!("{}", s.first_word());
}
```

5 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`) wie ein normaler Wert verwendet werden kann. Dies kann zu schwerwiegenden Laufzeitfehlern führen.

5.1 `Option<T>` statt `null`

Rust kennt kein `null`. Stattdessen wird die An- und Abwesenheit eines Wertes mit der Enumeration `Option<T>` gelöst, welche mit Javas `Optional` vergleichbar und 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"),
}
```

Das Codebeispiel `pattern-matching` demonstriert eine weitere Verwendung von `Option<T>`.

5.2 Erzwungene Handhabung aller Fälle

`match` ist vergleichbar mit `switch` in Java. 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.

Gibt es Fälle, auf die man im jeweiligen Kontext nicht reagieren möchte, kann man diese einfach dem *Einheitswert* () zuordnen:

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

Da diese Syntax etwas umständlich ist, können Fälle, die nicht von Interesse sind, mit den Platzhalter _ zusammengefasst werden. Da dieser auf alle Werte passt, muss er als letzter Arm aufgeführt sein:

```
match guess.cmp(&secret_number) {
    Ordering::Equal => println!("Right guess!"),
    _ => (),
}
```

5.3 Vereinfachung mit `if/let`

Da bei vielen Operationen nur auf eine einzige Art von Ergebnis reagiert werden soll, bietet Rust ein kompakteres Konstrukt für solche Fälle an. Mit `if/let` kann der vorherige `match`-Ausdruck folgendermassen umgeschrieben werden:

```
if let Ordering::Equal = guess.cmp(&secret_number) {
    println!("Right guess!");
}
```

Der Nachteil an diese Konstrukt ist, dass der Compiler nicht prüft, ob alle möglichen Fälle abgedeckt werden.

6 Concurrency mit Threads

Rust versteht sich als Low-Level-Programmiersprache. Ein Thread in Rust wird beispielsweise 1:1 auf einen Betriebssystem-Thread abgebildet. (Dies lässt sich auf Unix-Systemen mit `top -H` nachprüfen.) In Kombination mit einer High-Level-API lassen sich in Rust aber dennoch mit wenig Aufwand (und wenig Code) elegent Probleme nebenläufig lösen.

Ein Thread wird über die assoziierte Funktion (vgl. statische Methode) `spawn` gestartet, indem als Parameter eine anonyme Funktion mitgegeben wird. Zurück erhält man einen `JoinHandle<T>`, wobei die Typangabe dem Rückgabetyp der übergebenen anonymen Funktion entspricht:

```
let handle: thread::JoinHandle<String> = thread::spawn(|| {
    String::from("I'm a thread.")
});
```

Die beiden Pipes (`||`) nach `spawn` bezeichnen eine leere Parameterliste für die anonyme Funktion, die dem Thread mitgegeben wird. Würde die Thread-Funktion Werte vom umschliessenden Gültigkeitsbereich verwenden, müsste deren Ownership zunächst mit `move` übergeben werden:

```
thread::spawn(move || {
    // use values from surrounding scope
});
```

Auf die Abarbeitung vom Thread kann mit der Methode `join()` von `handle` gewartet werden. Diese Methode gibt sogleich den Wert zurück, der von der anonymen Thread-Funktion zurückgegeben wird. Dieser wird allerdings nicht direkt, sondern in ein `Result<T>` verpackt zurückgegeben. Ein `Result<T>` ist vergleichbar mit einer `Option<T>`, mit dem Unterschied, dass die Varianten `Ok` und `Err` heissen, und die zweite Variante einen Fehler beinhaltet. Das `Result` kann mittels `match` entpackt werden:

```
match handle.join() {
    Ok(v) => println!("The thread says: '{}'", v),
```

```
    Err(e) => panic!(e),
}
```

Der zurückgegebene Wert wird in eine zusätzliche String-Nachricht verpackt ausgegeben. Im Fehlerfall wird das Problem mit `panic!` weiter nach oben delegiert – vergleichbar mit `throw` in Java.

Auf der Basis von Threads unterstützt Rust verschiedene Concurrency-Modelle: Shared State und Message Passing.

6.1 Shared State mit Mutex

Bei Shared-State-Concurrency greifen mehrere Threads wechselseitig auf gemeinsame Speicherbereiche zu. Mittels pessimistischem Locking via Mutex wird sichergestellt, dass sich die einzelnen Threads dabei nicht in die Quere kommen. Möchte man etwa einen Zähler von mehreren Threads hochzählen lassen, bietet Rust dafür einen atomaren Zähler (`Arc`: atomic reference counter). Dieser wird mit einem `Mutex` ausgestattet, der wiederum einen Variable schützt, die hier als Integer-Variable mit dem Wert `0` angegeben wird:

```
let counter = Arc::new(Mutex::new(0));
```

Ein Thread kann den Zähler nun folgendermassen erhöhen:

```
thread::spawn(move || {
    // do something before
    {
        let mut c = counter.lock().unwrap();
        *c += 1;
    }
    // do something else afterwards
});
```

Hier wird `move` benötigt, damit der Thread auf den Counter zugreifen kann. Dieser bzw. dessen `Mutex` wird per `lock()` gesperrt. (Die Methode `unwrap()` gibt den Rückgabewert von `lock()` zurück. Im Fehlerfall würde der Fehler mit `panic` weitergereicht. Das ist kompakter als ein `match`-Konstrukt.) Nach der Erhöhung des Zählers wird der `Mutex` *nicht* explizit, sondern implizit am Blockende freigegeben. Aus diesem Grund wurde hier ein zusätzlicher innerer Block eingeschoben. Würde der Thread im gleichen Block nach der Erhöhung des Zählers noch weitere Arbeit ausführen, bliebe die Sperre solange aufrechterhalten.

Das Codebeispiel `mutex` zeigt eine beispielhafte Anwendung.

6.2 Message Passing mit Channels

Eine Alternative zur fehleranfälligen Manipulation geteilter Speicherbereiche ist das Message Passing, wobei mehrere Threads über Channels Informationen untereinander austauschen. Das Prinzip ist mit Unix-Pipes vergleichbar, und die Rust-Implementierung ist von

Go inspiriert. (Die Go-Entwickler beziehen sich dabei auf Tony Hoares *Communicating Sequential Processes*. Die andere bekannte Erfindung von Tony Hoare – `null` – findet sich dabei auch in Go wieder, jedoch nicht in Rust.)

Eine mögliche Channel-Implementierung in Rust ist das Modul `mpsc` (Multi Producer, Single Consumer), das auf einer FIFO-Queue basiert. Channels haben zwei Teile, einen Transmitter (Sender) und einen Receiver, welche man bei der Erstellung eines Channels als Tupel erhält und idiomatisch mit `tx` (Transmitter) und `rx` (Receiver) bezeichnet:

```
let (tx, rx) = mpsc::channel();
```

Jeder Thread muss seine eigene Kopie vom Transmitter anlegen, damit er auf den Channel schreiben kann:

```
let tx_copy = mpsc::Sender::clone(&tx);
tx_copy.send(1).unwrap();
```

Im Hauptthread (oder einem beliebigen anderen Thread) kann der Channel über eine Iteration konsumiert werden:

```
for messages in rx {
    counter += message; // getting increments from threads
}
```

Die Schleife läuft solange, bis alle Transmitter geschlossen sind. Dies geschieht explizit: in jedem Thread mit `drop(tx_copy)` bzw. im Hauptthread mittels `drop(tx)`.

Das Codebeispiel `chans` implementiert das semantisch gleiche Programm wie das Beispiel `mutex`, verwendet dazu jedoch einen Channel anstelle eines Mutexes. Die Implementierung mit dem Channel ist dabei etwas kürzer und eleganter.

7 Fazit

7.1 Technisches Team-Fazit

Rust bietet einige sehr interessante Konzepte wie zum Beispiel Ownership und Traits. Diese Konzepte können sehr nützlich sein, aber bringen auch den einen oder anderen Stolperstein mit sich. So ist es beispielsweise sehr schwierig eine Linked-List für einen Stack zu implementieren, da sich der Compiler ständig wegen Ownership-Problemen beschwert.

Das Pattern Matching ist ein mächtiger Mechanismus zur Behandlung unterschiedlicher Fälle. Verglichen mit dem Ownership-Konzept lässt einem der Compiler hier aber mehr Freiheiten. Das Concurrency-Model von Rust schafft es, Low-Level-Threads mit High-Level-APIs zu kombinieren. Nebenläufiger Code lässt sich effizient *und* schön implementieren.

Das gesamte Tooling für die Sprache ist sehr gut, und der Compiler ist ziemlich intelligent. Die Fehlermeldung des Compiler sind in der allermeisten Fällen sehr nützlich und beschrei-

ben das Problem sehr genau. Manchmal wird sogar ein Vorschlag angezeigt, wie man es machen soll. Dadurch entsteht meistens recht guter Code.

Robert C. Martin vertritt die These, dass sich neue Programmierparadigmen dadurch auszeichnen, dass sie bestehende Paradigmen *einschränken*. (Strukturierte Programmierung: Einschränkung von Sprüngen, d.h. kein `goto`; Objektorientierte Programmierung: Einschränkung von Funktionszeigern; Funktionale Programmierung: Einschränkung von Zustandsänderungen.) Rust überträgt die Einschränkung von Zustandsänderungen (und Datenzugriff) auf die strukturierte und parallele Programmierung – und eliminiert so eine ganze Fehlerklasse (Race Conditions).

7.2 Persönliches Fazit Patrick

Rust verlangt dem Programmierer einiges ab. Mit systematischem Lernen vor dem Einsatz dürfte man schneller vorwärts kommen als mit «Learning by Doing», da man wichtige Konzepte (gerade Ownership) zuerst verstehen muss, bis man überhaupt kompilierbaren Code schreiben kann. Ich freue mich sehr über die Renaissance der kompilierten Programmiersprachen, denn ein Compiler kann viele Fehler ausräumen, die bei Bytecode- und Skriptsprachen erst zur Laufzeit auftauchen. Der Rust-Compiler ist hier besonders stark. Persönlich bin ich zwischen Rust und Go hin- und hergerissen, wobei Go bisher mein klarer Favorit war. Nach meiner ersten ernsthaften Annäherung an Rust möchte ich beide Programmiersprachen weiterverfolgen, und hoffe, dass ich einmal beide in einem produktivem Umfeld verwenden kann.

7.3 Persönliches Fazit Lukas

Die Sprache ist für mich eine sehr interessante Alternative zu C. Für kleinere CLI-Tools finde ist die Sprache perfekt. Das Ökosystem der Sprache ist sehr lebending, was man beispielsweise daran sieht, dass während dem wir und mit der Sprache befasst haben, eine neu Version veröffentlicht und auch die Webseite erneuert wurde. Ich empfehle die Sprache jedem weiter der eine Alternative zu C sucht oder einfach ein paar spezielle Konzepte kennenlernen möchte.