Hochschule Esslingen
University of Applied Sciences

Fakultät Informationstechnik
im Studiengang Softwaretechnik und Medieninformatik

Bachelorarbeit

Entwurf und Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens in Rust

Michael Watzko

Sommersemester 2018
14.02.2018 - 22.06.2018

Erstprüfer: Prof. Dr. rer. nat. Dipl.-Inform. Manfred Dausmann
Zweitprüfer: M. Sc. Kevin Erath

IT DESIGNERS GRUPPE

Firma: IT Designers GmbH
Betreuer: M. Sc. Kevin Erath

[git] • master @ 8de7069 • 15. März 2018

Sperrvermerk

Vermutlich relevant weil Details eines Forschungsprojekts?

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 15. März 2018 _____

Michael Watzko

Danksagungen

„This occasionally happens in Rust: there is a period of intense arguing with the compiler, at the end of which the code looks rather nice, as if it had been a breeze to write, and runs beautifully.“

– Jim Blandly und Jason Orendorff in Programming Rust

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Tesla Autos einen allgemeinen Bekanntheitsgrad erreicht. Damit ein Auto selbstständig fahren kann, müssen erst viele Hürden gemeistert werden. Dazu gehört zum Beispiel das Spur halten, das richtige Interpretieren von Verkehrsschildern und das Navigieren durch komplexe Kreuzungen.

Bevor ein autonomes Fahrzeug Entscheidungen treffen kann, benötigt es ein möglichst genaues Model seines Umfelds. Hierzu werden von verschiedene Sensoren wie Front-, Rück- und Seitenkameras und Abstandssensoren Informationen gesammelt und ausgewertet. Aber vielleicht kann ein Auto nicht immer selbständig genügend Informationen zu seinem Umfeld sammeln?

Externe Sensorik könnte Informationen liefern, die das Auto selbst nicht erfassen kann. Ein viel zu schneller Radfahrer hinter einer Hecke in einer unübersichtlicher Kreuzung? Eine Lücke zwischen Autos, die ausreichend groß ist, um einzufahren ohne zu bremsen? Die nächste Ampel wird bei Ankunft rot sein, ein schnelles und Umwelt belastendes Anfahren ist nicht nötig? Ideen gibt es zuhauf.

Aber was ist, wenn das System aussetzt? Die Antwort hierzu ist einfach: das Auto muss immer noch selbstständig agieren können, externe Systeme sollen nur optionale Helfer sein. Viel schlimmer ist es dagegen, wenn das unterstützende System falsche Informationen liefert. Eine Lücke zwischen Autos, wo keine ist; eine freie Fahrbahn, wo ein Radfahrer fährt; ein angeblich entgegenkommendes Auto, eine unnötig Vollbremsung, ein Auffahrunfall. Ein solches System muss sicher sein – nicht nur vor Hackern. Es muss funktional sicher sein, Redundanzen und Notfallsysteme müssen jederzeit greifen.

Aber was nützt die beste Idee, die ausgeklügelte Strategie gegenüber einer undefinierten Situation in der verwendeten Programmiersprache? Wenn nur ein einziges mal vergessen wurde, einen Rückgabewert auf den Fehlerfall zu prüfen? Was nützt es, wenn Strategien für das Freigeben von Speicher in Notfallsituationen einen Sonderfall übersehen haben? Das System handelt total unvorhersehbar.

Was wäre, wenn es eine Programmiersprache geben würde, die so etwas nicht zulässt; die fehlerhaften Strategien zur Compilezeit findet und die Compilation stoppt. Die trotz erzwungener Sicherheitsmaßnahmen, schnell und echtzeitnah reagieren kann und sich nicht

vor Geschwindigkeitsvergleichen mit etablierten, aber unsicheren Programmiersprachen, scheuen muss?

Diese Arbeit soll zeigen, dass Rust genau so eine Programmiersprache ist und sich für sicherheitsrelevante, hoch parallelisierte und echtzeitnahe Anwendungsfälle bestens eignet.

1.2 Projektkontext

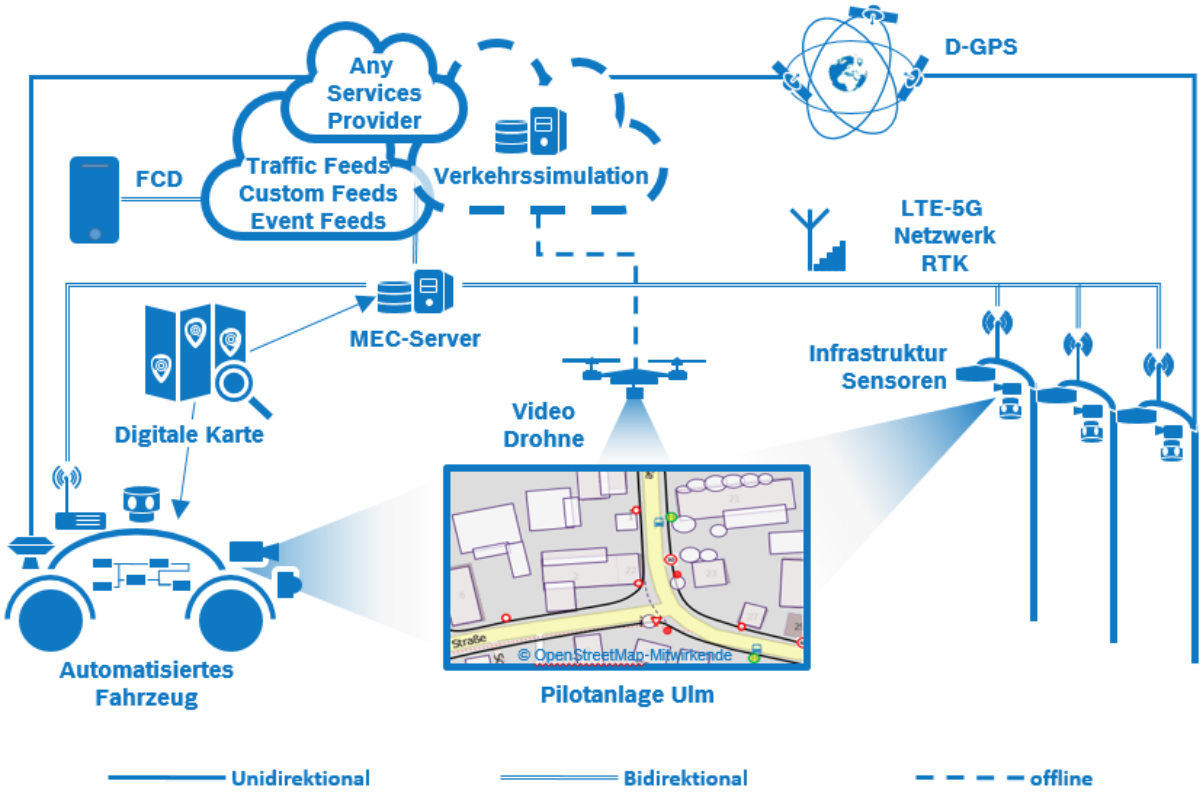

Abbildung 1.1: Übersicht über das Forschungsprojekt [**mec:home**]

Quelle: https://www.uni-due.de/~hp0309/images/Arch_de_V1.png (modifiziert)

Diese Abschlussarbeit befasst sich mit dem Kommunikationsserver von MEC[1]-View. Das MEC-View Projekt wird durch das BMWi[2] gefördert und befasst sich mit der Thematik autonom fahrender Fahrzeuge. Es soll erforscht werden, ob und in wie weit eine durch externe Sensorik geleistete Unterstützung nötig und möglich ist, um in eine Vorfahrtstraße autonom einzufahren.

Das Forschungsprojekt ist dabei ein Zusammenschluss mehrerer Unternehmen mit unterschiedlichen Themengebieten. Die IT-Designers Gruppe beschäftigt sich mit der Implementation des Kommunikationsservers, der auf der von Nokia zur Verfügung gestellten Infrastruktur im 5G Mobilfunk als MEC Server betrieben wird. Erkannte Fahrzeuge und andere Verkehrsteilnehmer werden von den Sensoren von Osram via Mobilfunk an den Kommunikationsserver übertragen. Der Kommunikationsserver stellt diese Informationen

[1] Mobile Edge Computing
[2] Bundesministerium für Wirtschaft und Energie

dem Fusionsalgorithmus der Universität Ulm zur Verfügung und leitet das daraus gewonnene Umfeldmodell an die hochautomatisierten Fahrzeuge von Bosch und der Universität Ulm weiter. Durch hochgenaue, statische und dynamische Karten von TomTom und den Fahrstrategien von Daimler soll das Fahrzeug daraufhin autonom in die Kreuzung einfahren können.

1.3 Zielsetzung

Das Ziel ist es, eine alternative Implementierung des MEC-View Servers in Rust zu schaffen. Durch die Garantien (Abschnitt 2.22) von Rust wird erhofft, dass der menschliche Faktor als Fehlerquelle gemindert und somit eine fehlertolerantere und sicherere Implementation geschaffen werden kann.

Eine Ähnlichkeit in Struktur und Architektur zu der bestehender C++ Implementation ist explizit nicht vonnöten. Eventuelle Spracheigenheiten und einzigartige Features von Rust sollen im vollen Umfang genutzt werden können, ohne durch Auferzwungene und unpassende Architekturmuster benachteiligt zu werden. Es ist erwünscht eine kompetitive Implementation in Rust zu schaffen.

1.4 Aufbau der Arbeit

Diese Arbeit ist im wesentlichen in die folgenden Themengebiete aufgeteilt: Grundlagen, Anforderungs- und Systemanalyse, Systementwurf und Implementation und Auswertung.

Im Themengebiet Grundlagen sollen wesentliche Bestandteile dieser Arbeit erläutert und erklärt werden. Hierzu zählt zum einen die Programmiersprache Rust in ihrer Entstehungsgeschichte (siehe Abschnitt 2.1), Garantien und Sprachfeatures (siehe Abschnitt 2.22). Zum anderen die hochperformante, serverbasierte Kommunikationsplattform mit ihren Protokollen (ab Kapitel 3) und dem Systemkontext in dem diese betrieben wird.

In der Anforderungs- und Systemanalyse wird der Kontext in dem das System betrieben werden soll genauer betrachtet. Umzusetzende funktionale und nicht-funktionale Anforderungen werden aufgestellt, sowie eine Übersicht von Systemen mit denen das System interagiert wird.

Das Themengebiet Systementwurf und Implementation befasst sich mit dem theoretischen und praktischen Lösen der im vorherigen Kapitel aufgestellten Anforderungen. Aufgrund der Tatsache, dass es sich hierbei um eine alternative Implementation handelt, wird zur bestehenden C++ Implementation Bezug genommen. Auf architektonische Unterschiede im Systementwurf, die sich aufgrund von Sprach- und Bibliotheksunterschiede, werden hier genauer beschrieben.

Zuletzt wird eine Auswertung der Implementation aufgezeigt.

TODO: entsprechend zu aktualisieren

2 Die Programmiersprache Rust

Rust hat als Ziel, eine sichere (siehe Abschnitt 2.22) und performante Systemprogrammiersprache zu sei. Abstraktionen sollen die Sicherheit, Lesbarkeit und Nutzbarkeit verbessern aber keine unnötigen Performanceeinbußen verursachen (siehe Unterabschnitt 2.22.6).

Aus anderen Programmiersprachen bekannte Fehlerquellen – wie „dangling pointers“, „double free“ oder „memory leaks“ – werden durch strikte Regeln und mit Hilfe des Compilers verhindert (Abschnitt 2.22). Im Gegensatz zu Programmiersprachen, die dies mit Hilfe ihrer Laufzeitumgebung[1] sicherstellen, werden diese Regeln in Rust durch eine statische Lebenszeitanalyse (Abschnitt 2.18) und mit dem Eigentümerprinzip (Abschnitt 2.19) bei der Compilation überprüft und erzwungen.

Diese erlaubt Rust eine zur Laufzeit hohe Ausführgeschwindigkeit zu erreichen. Das Eigentümerprinzip (siehe Abschnitt 2.19) und die Markierung durch von Datentypen durch Merkmale (siehe Abschnitt 2.9) vereinfacht es, nebenläufige und sichere Programme zu schreiben.

Rust hat in den letzten Jahren viel an Beliebtheit gewonnen und ist 2018 das dritte Jahr in Folge als die beliebteste Programmiersprache in einer Umfrage auf Stack Overflow gewählt worden [**rust:stack_overflow:mose_loved**]. Rust scheint dem Anspruch, eine sichere und performante Programmiersprache zu sein, gerecht zu werden:

> *„Again, Rust guides you toward good programs“* [**rust:orly_programming**]

> *„[..]Leute, die [..] sichere Programmierung haben wollen, [..] können das bei Rust haben, ohne [..] undeterministischen Laufzeiten oder Abstraktionskosten schlucken zu müssen. “* [**rust:fefe**]

> *„[..] Rust makes it safe, and provides nice tools“* [**rust:c_is_hostile_mena**]

> *„Rust hilft beim Fehlervermeiden“* [**rust:c_is_hostile_golem**]

> *„Rust is [..] a language that cares about very tight control“* [**rust:tight_control**]

[1] u.a. Java Virtual Maschine (JVM), Common Language Runtime (CLR)

2.1 Geschichte

In 2006 begann Graydon Hoare die Programmiersprache Rust in seiner Freizeit als Hobbyprojekt zu entwickeln [**rust:faq**]. Als Grund nannte er seine Unzufriedenheit mit der Programmiersprache C++, in der es sehr schwierig sei, fehlerfreien, speichersicheren und nebenläufigen Programmcode zu entwickeln. Zudem beschrieb er C++ als „ziemlich fehlerträchtig“ [**rust:heise_interview_graydon**].

Auch Federico Mena-Quintero – Mitbegründer des GNOME-Projekts [**rust:gnome:federico**] – äußerte in einem Interview mit Golem im Juli 2017 seine Bedenken an der Verwendung der „feindseligen“ Sprache C [**rust:c_is_hostile_golem**]. In Vorträgen vermittelt er seither, wie Bibliotheken durch Implementationen in Rust ersetzt werden können [**rust:c_is_hostile_mena**].

Ab 2009 begann Mozilla die Weiterentwicklung finanziell zu fördern, als einfache Tests und die Kernprinzipien demonstriert werden konnten. Die Entwicklung der Programmiersprache, des Compilers, des Buchs, von Cargo, von crates.io und von weiteren Bestandteilen findet öffentlich einsehbar auf GitHub[2] unter `https://github.com/rust-lang` statt und wird nicht ausschließlich von Mozilla Angestellten koordiniert. Dadurch kann sich jeder an Diskussionen oder Implementation beteiligen, seine Bedenken äußern oder Verbesserungen vorschlagen.

Durch automatisierte Tests (siehe Abschnitt 2.13) in Kombination mit drei Veröffentlichungskanälen („relese“, „stable“ und „nightly“) und „feature gates“ (siehe Unterabschnitt 2.2.2) wird die Stabilität des Compilers und die der Standardbibliothek (Abschnitt 2.17) gewährleistet.

Rust ist wahlweise unter MIT oder der Apache Lizenz in Version 2 verfügbar [**rust:copyright**].

2.2 Anwendungsgebiet

Das Ziel von Rust ist es, das Designen und Implementieren von sicheren und nebenläufigen Programmen möglich zu machen. Gleichzeitig soll der Spagat geschaffen werden, nicht nur ein sicheres aber lediglich theoretisches Konstrukt zu sein, sondern in der Praxis eine Anwendung zu finden. Als Beweis könnte hierbei auf die Umstellung von Firefox auf Rust und Servo – ein minimaler Webbrowser komplett in Rust geschrieben – verwiesen werden [**rust:faq**].

[2] Plattform zum Hosten von git-Repositories inklusive eingebautem Issue-Tracker und Wiki. Änderungen an Quellcode können vorgeschlagen, und durch die Projektverantwortlichen übernommen werden. Bietet auch die Möglichkeit eine kontinuierlichen Integrationssoftware einzubinden, um automatisierte Tests auf momentanen Quellcode und auch für Änderungen auszuführen. Eine vorgeschlagene Änderung kann somit vor Übernahme auf Kompatibilität überprüft werden.

Interessant ist eine Diskussion von 2009, bei der „sicher aber nutzlos“ und „unsicher aber brauchbar“ Gegenübergestellt wurde. Programmiersprachen scheinen auf der Suche nach dem nicht existierende „Nirvana“ zu sein, das sowohl sichere als auch brauchbare Programmierung verspricht [**rust:infoq:null**]. Rust möchte dieses Nirvana gefunden haben.

2.2.1 Kompatibilität

Da Rust den LLVM[3]-Compiler nutzt, erbt Rust auch eine große Anzahl der Zielplattformen die LLVM unterstützt. Die Zielplattformen sind in drei Stufen unterteilt, bei denen verschieden stark ausgeprägte Garantien vergeben werden. Es wird zwischen

- „Stufe 1: Funktioniert garantiert“ (u.a. X86, X86-64),
- „Stufe 2: Compiliert garantiert“ (u.a. ARM, PowerPC, PowerPC-64) und
- „Stufe 3“ (u. a. Thumb (Cortex-Microcontroller))

unterschieden [**rust:platform_support**]. Diese Unterscheidung wirkt sich auch auf die Stabilisierungsphase und Implementation neuer Funktionen aus (Beispiel „128-bit Integer Support“ [**rust:github:128bit_integer**]).

2.2.2 Veröffentlichungszyklus

Es stehen Versionen in drei verschieden Veröffentlichungskanälen zur Verfügung:

- **nightly**: Version die einmal am Tag mit dem aktuellen Stand des Quellcodes gebaut wird. Experimentelle und nicht fertige Features sind hier zwar enthalten, aber hinter „feature gates“ versteckt. Diese „Tore“ können durch entsprechende Attribute (siehe Abschnitt 2.12) geöffnet werden, so ermöglicht (`#[feature(const_fn)]` die Definition von Konstante Funktionen (Stand 15. März 2018).
- **beta**: Alle sechs Wochen wird die aktuellste Nightly zur Beta befördert und es werden nur noch Fehler aus dieser Version getilgt. Dieser Prozess könnte auch als Reifephase bezeichnet werden.
- **stable**: Nach sechs Wochen wird die aktuellste Beta zur Stable befördert und veröffentlicht. Gleichzeitig wird auch eine neue Beta veröffentlicht.

[3] Früher „Low Level Virtual Machine“ [**wiki:llvm**], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [**llvm:home**]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [**llvm:features**].

2.2.3 Ökosystem

Mit Rust wird nicht nur eine Programmiersprache, sondern auch ein umfassendes Ökosystem angeboten.

Cargo ist vermutlich das größte angebotene Werkzeug. Es löst Abhängigkeiten auf, indem es auf das öffentliche Verzeichnis unter `https://crates.io` zurückgreift und diese entsprechend herunterlädt und compiliert. Zum jetzigen Zeitpunkt (15. März 2018) sind über 14.000 Crates öffentliche erreichbar und nutzbar. Zudem wird durch Crago eine *Cargo.toml* verlangt, in der Metainformationen einer Crate hinterlegt sind. Dies Umfasst u.a. Name, Version, Autor, Lizenz und Abhängigkeiten.

Eine Crate kan von jedem veröffentlicht werden, insofern derjenige ein GitHub-Konto besitzt, der Name der Crate noch nicht vergeben ist und der Programmcode compiliert. Die API-Dokumentation der jeweiligen Crate wird dabei automatisiert auf `https://docs.rs` veröffentlicht.

Unter `https://www.rust-lang.org` ist die Website von Rust erreichbar und unter `https://doc.rust-lang.org` sowohl die API-Dokumentation der Standardbibliothek als auch as Hauseigene Rust Buch in Version 1 und 2. Die Entwicklung findet dagegen auf GitHub unter `https://github.com/rust-lang` statt.

Kleine Testprogramme und Experimente können auf dem „Spielplatz“ unter `https://play.rust-lang.org` compiliert und ausgeführt werden, ohne lokal etwas zu installieren.

2.3 Aufbau eines Projektverzeichnisses

Der Aufbau eines Rust Projektverzeichnis kann zwischen zwei verschiedenen Arten differenziert werden. Zum einen gibt es den klassische Aufbau, in dem lediglich der Programmcode liegt und der Compiler direkt aufgerufen und parametrisiert wird. Zum anderen wird der Aufbau als Crate (siehe Unterabschnitt 2.3.2) empfohlen, da dadurch Abhängigkeiten automatisch aufgelöst werden können aber auch Metainformationen bezüglich des Autors, der Version und der Abhängigkeiten hinterlegt werden müssen. Ein klassischer Aufbau ist dagegen nur selten anzutreffen.

2.3.1 Klassisch

Das Quelldatei-Verzeichnis sollte entweder eine *main.rs* für Ausführbare Programme oder eine *lib.rs* für Bibliotheken enthalten. Während der Paketmanager Cargo eine solche Benennung als Standardkonvention erwartet, kann bei manueller Nutzung des Compilers auch ein anderer Name für die Quelldatei vergeben werden.

```
src/
|-- main.rs
|-- functionality.rs
|-- module/
    |-- mod.rs
    |-- functionality.rs
    |-- submodule/
        |-- mod.rs
        |-- functionality.rs
```

Listing 2.1: Verzeichnisstruktur des Quelltext-Verzeichnisses

Der Compiler startet in der Wurzeldatei und lädt weitere Module, die durch `mod module;` gekennzeichnet sind (ähnlich `# include "module.h"` in C/C++). Ein Modul kann dabei eine weitere Quelldatei oder ein ganzes Verzeichnis sein. Ein Verzeichnis wird aber nur als gültiges Modul interpretiert, wenn sich eine *mod.rs* Datei darin befindet. Um Datentypen und Funktionen aus einem Modul nutzen zu können, ohne dessen kompletten Pfad jedes mal auszuschreiben, müssen sie durch zum Beispiel `use module::functionality::Data;` in dem aktuellen Namensraum bekannt gemacht werden.

Wie bereits angedeutet, wird in Rust nicht eine „Klasse“, Datenstruktur oder Aufzählung pro Datei erwartet, sondern eine Quelldatei entspricht einem Modul. Diese Umfasst in vielen fällen wenige aber mehrere Datenstrukturen, zugehörige Aufzählung und Fehlertypen.

2.3.2 Als Crate

Eine „Crate“ (dt. Kiste/Kasten) erweitert den klassischen Aufbau um eine *Cargo.toml* Datei, in der Metainformationen zum Projekt hinterlegt werden. Durch die Benutzung des Werkzeugs „Cargo“ (dt. Fracht/Ladung) können Abhängigkeiten automatisch aufgelöst, heruntergeladen und compiliert werden.

```
crate/
|-- Cargo.toml
|-- src/
    |-- ...
```

Listing 2.2: Vereinfachte Verzeichnisstruktur einer „crate“

Eine Crate kann entweder ein ausführbares Programm oder eine Bibliothek sein. Davon abhängig is die Wurzeldatei *src/main.rs* (für ein ausführbares Programm) oder *src/lib.rs* (für eine Bibliothek). Mit dem erzeuge einer Crate (`cargo new --bin meinProg` bzw. `cargo new --lib meineBib`) wird auch gleichzeitig git[4] für das Verzeichnis initialisiert.

[4] (dt. Blödmann) ist eine Software zur Versionierungs von Quelldateien, entwickelt von Linus Torvalds 2005. TODO: cite

2.4 Hello World

```
1 fn main() {
2     println!("Hello World");
3 }
```

Listing 2.3: „Hello World“ in Rust

Der Programmcode in Listing 2.3 gibt auf der Konsole `Hello World` aus. Das `fn` die Funktion `main` definiert und diese der Startpunkt des Programms ist, wird vermutlich wenig überraschend sein. Viel überraschender ist vermutlich eher das Ausrufezeichen in Zeile 2, da es auf den ersten Blick dort nicht hingehören sollte. In Rust haben Ausrufezeichen und Fragezeichen besondere Bedeutungen, weswegen die Verwendung in Zeile 2 trotzdem richtig ist.

Die Bedeutung des Fragezeichens dient zum schnelleren Auswerten von `Rusult<_, _>`-Werten und wird in Unterabschnitt 2.22.3 genauer erklärt. Das Ausrufezeichen kennzeichnet, dass der ansonsten augenscheinliche Funktionsaufruf tatsächlich ein Aufruf einer Makrofunktion ist.

Eine Funktion `println` gibt es nicht, auch keine aus C erwarteten Funktionen wie `printf`, `fputs` oder `sprintf`. Eine Ausgabe erfolgt durch das `println!` Makro, welches eine String durch Nutzung des `format!` Makros formatiert und erstellt. Daraufhin wird das `writeln!` Makro verwendet, um die formatierte Zeichenkette auf die Standardausgabe zu schreiben.

2.5 Einfache Datentypen

Die Datentypen in Rust sind im wesentlichen die üblichen Verdächtigen: `bool` für boolische Ausdrücke; `char` für ein einzelnes Unicode Zeichen; `str` für eine Zeichenkette; `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, (bald `u128`, `i128` [**rust:github:128bit_integer:rfc**]) und `usize`, `isize` für ganzzahlige Werte; `f32`, `f64` für Fließkommazahlen in einfacher und zweifacher Präzision; Arrays und Slices [**rust:book:primitives**].

Ganzzahlige Datentypen mit einem führenden `u` sind vorzeichenlos („unsigned“), vorzeichenbehaftete Datentypen („signed“) sind dagegen mit einem `i` gekennzeichnet. Fließkommazahlen sind stattdessen mit einem führenden `f` („floating point“) gekennzeichnet. Die darauf folgende Zahl gibt die Anzahl der Bits wieder, die der Datentyp groß ist. Die einzige Ausnahme sind die ganzzahligen Datentypen `usize` und `isize`, da diese immer so groß sind, wie die Architektur der Zielplattform. Für die Indexierung eines Arrays oder einer Slice würden andere Datentypen, mit einer fest definierten Größe, keinen Sinn ergeben, da das Maximum an adressierbaren Elementen von der Architektur der Zielplattform abhängig ist.

Durch dieses Schema bei der Bezeichnung der Datentypen wird eine Verwirrung wie zum Beispiel in C unterbunden, wo die primitiven Datentypen (`short` , `int` , `long` , ..) keine definierte Größe haben, sondern dies abhängig vom eingesetzten Compiler und der Zielplattform ist [**deitel2013c**]. Erst ab C99 wurden zusätzliche, aber optionale, ganzzahlige Datentypen mit festen Größe definiert [**goll2014c**].

Konstanten können in Rust direkt einem Datentyp zugewiesen werden, indem dieser angehängt wird: `4711u16` ist vom Datentyp `u16` . Unterstriche dürfen an beliebiger stelle Ziffern trennen, um die Lesbarkeit zu erhöhen: `1_000_000_f32` . Eine Schreibweise in Binär (`0b0000_1000_u8`), in Hexadezimal (`0xFF_08_u16`) oder in Oktal (`0o64_u8`) ist auch möglich. Konstante Zeichen und Zeichenketten können auch automatisch durch ein vorangestelltes *b* in Bytes gewandelt werden: `b'b'` entspricht `0x62_u8` und `b"abc"` entsricht `&[0x61_u8, 0x62_u8, 0x63_u8]` .

Arrays haben immer eine zur Compilezeit bekannte Größe und müssen auch immer mit einem Wert initialisiert werden (siehe Unterabschnitt 2.22.1). Dynamische Arrays auf dem Stack gibt es (noch? [**rust:github:alloca**]) nicht, stattdessen wird auf die Vektor Implementation der Standardbibliothek verwiesen (siehe Abschnitt 2.17). Die Notation für Arrays ist `[<Füllwert>; <Größe>]` , wobei die Größe ein konstanter Wert sein muss. `[0_u8; 128]` steht demnach für ein 128 Byte langes Array vom Datentyp `u8` , das mit 0-en initialisiert ist.

„Slices“ (dt. Scheiben/Stücke) bezeichnet Rust Referenzen auf Arrays oder auf Teilbereiche von Arrays und Slices. In einem so genannten „fat pointer“ wird der Startpunkt und die Größe der Slice gespeichert (siehe auch Abbildung 2.1). Der Compiler kann hierdurch einen Zugriff außerhalb einer Slice oder eines Arrays entweder zur Laufzeit, oder, falls möglich, zur Compilezeit verhindern. Ein Buffer-Overflow ist in Rust daher nicht möglich.

Die Notation ähnelt die eines Arrays, aber ohne Größenspezifikation: `&[<Datentyp>]` . Eine Slice kann zudem immer nur als eine Referenz angesprochen werden (siehe Abschnitt 2.6). Um eine Slice auf ein Array oder eine andere Slice zu erhalten, muss der Start- und Endindex des Teilbereiches angegeben werden. Falls kein Start- oder Endindex angegeben wird, wird das jeweilige Limit übernommen.

Folgendes Beispiel soll die Notation von Arrays und Slices Beispielhaft verdeutlichen:

```
1 fn main() {
2     let b : [u8; 10] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
3     for b in &b[2..5] {
4         print!("{}, ", b);
5     }
6 }
```

Listing 2.4: Beispiel eines Arrays und einer Slice

Das in Listing 2.4 gezeigte Programm, gibt auf der Konsole `2, 3, 4,` aus.

Variablen werden durch das `let` Schlüsselwort „gebindet“, das heißt, der Variable wird die Eigentümerschaft über den Wert zugewiesen. Ausnahmen können Datentypen mit dem Merkmal `Copy` bilden, da diese ein implizite Kopie erlauben (siehe Abschnitt 2.9). Anstatt eine Variable optional als unveränderlich zu kennzeichnen (`const` in C, `final` in Java), wird eine Variable in Rust optional als veränderlich gekennzeichnet (`mut`), während standardmäßig Variablen unveränderlich sind.

Lokale Typinferenz

Da Rust ein statisches Typensystem mit lokaler Typinferenz ist, muss der Datentyp einer Variable nicht notiert werden, sondern dieser wird automatisch erkannt. Dies gilt aber nur lokal, also innerhalb von Funktionen und Closures, für Parameterlisten und Rückgabewerte von Funktionen müssen die Datentypen explizit angegeben werden (siehe Abschnitt 2.7).

```
1 fn main() {
2   let a = 10_u32;  // Datentyp wird durch Konstante bestimmt
3   let b : u32 = a; // b muss vom Typ u32 sein
4   let c = b;       // c ist vom Typ u32, weil b u32 ist
5 }
```

Listing 2.5: Beispiel für lokale Typinferenz

2.6 Zusammengesetzen Datentypen

Die Programmiersprache Rust kennt neben den einfachen Datentypen (Abschnitt 2.5) weitere Möglichkeiten Daten zu organisieren:

- ein Tupel, das mehrere Werte namenlos zusammenfasst: `(f32, u8)` : `a.0 = 1.0_f32` ,
- eine Datenstruktur, die wie in C Datentypen namenbehaftet zusammenfasst: `struct Punkt { x: f32, y: f32 }` : `p.x = 1.0_f32` ,
- und Aufzählungen: `enum Bildschirm { Tv, Monitor, Leinwand }` .

Im Vergleich zu C kann ein Eintrag in einem `enum` gleichzeitig Daten wie eine Datenstruktur oder ein Tupel halten, oder lediglich einen Ganzzahlwert repräsentieren. Mit dem `type` Schlüsselwort können Aliase erstellt oder im Falle von FFI (siehe Abschnitt 2.23) aufgelöst werden: `type Vektor = (f32, f32);` Felder einer Struktur können zudem mit `pub` oder `pub(crate)` gekennzeichnet werden (siehe Abschnitt 2.10).

Seit Version 1.19 ist auch der Datentyp `union` in Rust verfügbar [**rust:v1.19**]. Eine `union` kann aber nur in `unsafe`-Blöcken verwendet werden, da der Compiler eine Ordnungsgemäße Nutzung nicht überprüfen kann. Für diese Abschlussarbeit hat der Datentyp aber keine Relevanz und wird daher nicht weiter erwähnt.

Referenzen

Auf alle Datentypen können Referenzen erstellt werden, um auf diese zuzugreifen, ohne sie zu konsumieren. In Rust spricht man dann oft davon, den Wert zu „leihen“, da sich der Eigentümer nicht ändert, sondern für den Gültigkeitsbereich der Referenz eine andere Variable auf den Wert verweist. Wie bei Variablen, wird zwischen Referenzen auf unveränderlichen und veränderlichen Werte unterschieden (siehe Abschnitt 2.19). Die Notation für Referenzen auf unveränderliche Werte ist `&<Datentyp>`. Erwartungsgemäß ist `&mut <Datentyp>` die Notation für Referenzen auf veränderliche Werte. Referenzen auf Referenzen sind möglich, auch mit unterschiedlichen Eine Manuelle Dereferenzierung einer Referenz ist in den allermeisten Fällen nicht nötig, sondern wird vom Compiler vorgenommen. In Fällen in denen dies nicht wie erwartet automatisch geschieht, kann eine Manuelle Dereferenzierung durch den `*`-Operator erzwungen werden.

2.7 Funktionen, Ausdrücke und Statements

Funktionen werden durch `fn` gekennzeichnet, gefolgt mit dem Funktionsnamen, der Parameterliste und zuletzt der Datentyp für den Rückgabewert. Selbst wenn kein expliziter Rückgabetyp angegeben wird, wird formal `()` zurück gegeben; `()` entspricht etwa `void` aus bekannten Programmiersprachen. Die Parameterliste unterscheidet sich von bekannten Programmiersprachen wie C und Java, indem zuerst der Variablenname und darauf folgend der Datentyp notiert wird.

```
1 fn add(a: f32, b: f32) -> f32 {
2     a + b
3 }
```

Listing 2.6: Beispiel einer Funktion

Obwohl in Zeile 2 von Listing 2.6 kein `return` zu sehen ist, wird trotzdem das Ergebnis der Addition zurückgegeben. Dies liegt daran, da in Rust vieles ein Ausdruck ist und somit einen Rückgabewerte liefert [**rust:book:statements**]. Auch ein if-else ist ein Ausdruck und kann einen Rückgabewert haben. Ein bedingter Operator (?:) is somit unnötig, da stattdessen ein if-else verwendet werden kann: `let a = if b { c } else { d };`. Auch eine Zeile mit einen Semikolon hat formal einen Rückgabewert: `()`.

2.8 Implementierung einer Datenstruktur

Zu einer Datenstruktur oder Aufzählung kann ein individuelles Verhalten implementiert werden. In dieser Kombination ähneln diese Konstrukte sehr einer Klasse aus bekannten objektorientierten Programmiersprachen, wie zum Beispiel Java, C# oder C++ (siehe auch Abschnitt 2.21).

Einen Konstruktor gibt es jedoch nicht; lediglich die Konvention, eine statische Funktion `new` stattdessen zu verwenden [**rust:book:constructors**]:

```
1  struct Punkt {
2      x: f32,
3      y: f32,
4  }
5
6  impl Punkt {
7      pub fn new(x: f32, y: f32) -> Punkt {
8          Punkt { x, y }
9      }
10 }
```

Listing 2.7: Punkt Datenstruktur mit einem „Konstruktor“

In seltenen Fällen wird auch `Default` implementiert (siehe Abschnitt 2.9), wodurch eine statische Funktion `default()` als Konstruktor ohne Parameter bereitgestellt wird.

Da eine Funktionsüberladung nicht möglich ist, soll bei weiteren Konstruktoren ein sprechender Name verwendet werden. Der `Vec<_>` der Standardbibliothek (siehe Abschnitt 2.17) bietet zum Beispiel zusätzlich `Vec::with_capacity(capacity: usize)` an, um einen Vektor mit einer bestimmten Größe zu initialisieren.

Für Funktionen können auch die Zugriffsmodifikatoren festgelegt werden (siehe Abschnitt 2.10).

2.9 Generalisierung durch Traits

Ähnlich wie Java oder C# bietet Rust durch einen eigenen Typ die Möglichkeit, ein gewünschtes Erscheinungsbild zu generalisieren, ohne gleichzeitig eine Implementation vorzugeben. Im Rust wird dieser Typ „Trait“ (dt. Merkmal) genannt.

Für Merkmale werden Funktionen in einem entsprechenden `trait <Name> { }` Block ohne Rumpf deklariert. Optional kann auch ein Standardrumpf implementiert werden, der bei einer Spezialisierung überschrieben werden darf. Auch auf ein Merkmal kann ein Zugriffsmodifikatoren gesetzt werden (siehe Abschnitt 2.10).

Die Implementation eines Merkmals wird für jeden Datentyp in einem separaten Block vorgenommen und entspricht der Notation `impl Merkmal for Datentyp { fn ... }`. Alternativ können Implementationen auch für ganze Gruppen von anderen Merkmalen vorgenommen werden: `impl<T> Merkmal for T where T: Clone { ... }` (entspricht: „implementiere `Merkmal` für alle, die `Clone`-bar sind“).

In Zukunft – oder jetzt in „nightly“ und hinter dem „feature gate“ `specialization` – wird es möglich sein, ein Standardverhalten für Gruppen zu implementieren und dieses später, für einen spezialisierten Fall, zu überschreiben [**rust:github:specialization**].

Merkmale unterscheiden sich in ihrer Handhabung gegenüber anderen Datentypen, da sie im allgemeinen keine bekannte Größe zur Compilezeit haben. Während dies in Programmiersprachen wie Java und C# automatisch durch die Darstellung abstrahiert und versteckt wird, hat ein Entwickler in Rust mehr Kontrolle über die Handhabung.

Dabei gibt es mehrere Vorgehensweisen:

- Die einfachste Art erfolgt über das Leihen mittels Referenz: `fn foo(bar: &Bar)` oder `fn foo(bar: &mut Bar)` – ein Unterschied zu anderen Datentypen ist nicht zu erkennen. Hierbei werden Funktionen aber dynamisch über eine „vtable“ aufgerufen, weswegen dies höhere Laufzeitkosten mit sich bringt. In Zukunft soll dieser Syntax eventuell durch `fn foo(bar: &dyn Bar)` und `fn foo(bar: &mut dyn Bar)` ersetzt werden, um auf den dynamischen Aufruf besser hinzuweisen [**rust:github:dyn**].
- Alternativ kann das Objekt, das das geforderte Merkmal implementiert, auf den Heap verschoben und anschließend davon die Eigentümerschaft übertragen werden. Dies ist möglich, da nach dem Verschieben auf den Heap die Größe der `Box` bekannt ist. Eine `Box` ist letztendlich nur ein Pointer auf einen Speicherbereich auf dem Heap. Ein Merkmal in einer `Box` wird „Trait-Object“ genannt und eine Funktionsdeklaration könnte so aussehen: `fn foo(bar: Box<Bar>)`.
- Die performanteste Alternative ist eine spezialisierte Funktion. Der Compiler dupliziert automatisch für jeden Datentyp die Funktion, setzt diesen ein und führt Optimierungen für den Datentyp durch (ähnlich einer Templateklasse in C++). In der Notation wird ein lokaler Typ deklariert, der als Bedingung ein oder Mehrere Merkmale implementiert haben muss: `fn foo<T: Bar>(bar: T)`.

Eine Deklaration `fn foo(bar: Bar)` für das Merkmal `Bar` ist nicht möglich, da zur Compilezeit eine eindeutige Größe nicht bekannt ist. Der zu reservierende Speicher für die Variable kann nicht bestimmt werden, weswegen eine Übergabe über den Stack nicht möglich ist.

Im folgenden werden oft anzutreffende und wichtige Merkmale aus der Standardbibliothek kurz erläutert:

- `Send`: Markiert einen Datentyp als zwischen Threads übertragbar. Automatisch für alle Datentypen implementiert, bei denen auch alle beinhalteten Datentypen von Typ Send sind. Manuelle Implementation ist nicht sicher [**rust:book:send_sync**].

`!Send` verhindert dagegen, dass ein Wert zu anderen Threads übertragen werden darf. Somit können ansonsten rein textuell beschriebene Beschränkungen, wie zum Beispiel für der OpenGL-Kontext, durch den Compiler überprüft und erzwungen werden.

- `Sync`: Markiert einen Datentype als zwischen Threads synchronisierbar, d.h. mehrere Threads dürfen gleichzeitig lesend darauf zugreifen. `!Sync` verbietet dies hingegen. Automatisch für alle Datentypen implementiert, bei denen auch alle beinhalteten Datentypen von Typ Sync sind. Manuelle Implementation ist nicht sicher [**rust:book:send_sync**].
- `Sized`: Verlangt eine zu Compilezeit bekannte Größe. `?Sized` erlaubt dagegen eine unbekannte Größe zur Compilezeit.
- `Copy`: Markiert einen Datentyp, der durch einfaches Speicherkopieren (etwa „memcpy“) vervielfacht werden kann. Verlangt, dass alle beinhalteten Datentypen auch `Copy` sind. Alle einfachen Datentypen sind bereits `Copy`.
- `Clone`: Markiert einen Datentyp der vervielfacht werden kann, dies jedoch nicht durch kopieren des Speichers möglich ist – zum Beispiel da der Referenzzähler von `Arc` oder `Rc` erhöht werden muss. Stellt die Funktion `clone` bereit, die dafür explizit aufgerufen werden muss. Verlangt für eine automatisierte Implementation, dass alle beinhalteten Datentypen auch `Clone` sind. Alle einfachen Datentypen sind bereits `Clone`.
- `Debug` und `Display`: Erzwingt die Implementation von Funktionen um einen Datentyp als Text darzustellen. Entweder mit möglichst vielen Zusatzinformationen (`Debug`) oder schön (`Display`). Verlangt für eine automatisierte Implementation, dass alle beinhalteten Datentypen auch `Debug` bzw `Display` sind.
- `Deafult`: Erzwingt die Implementation einer statische Methode `default()`, die wie ein leerer Standardkonstruktor von Java oder C# wirkt: Erzeugung einer neuen Instanz mit Standardwerten. Verlangt für eine automatisierte Implementation, dass alle beinhalteten Datentypen auch `Deafult` sind.
- `PartialEq`: Verlangt die Implementation einer Funktion um mit Instanzen des gleichen Typs verglichen werden zu können. Im Vergleich zu `Eq` erlaubt `PartialEq`, dass Typen keine volle Äquivalenzrelation haben. Dies ist zum Beispiel für den Vergleich von Fließkommazahlen wichtig, da laut IEE754 `Nan` ungleich zu allem ist, auch zu sich selbst (`Nan != Nan`) [**wiki:nan**][**rust:orly_programming**][**rust:doc:partialeq**].
- `Eq`: Erlaubt dem Compiler einen Vergleich auf Bit-Ebene durchzuführen, ungeachtet des Datentyps [**rust:doc:eq**].
- `PartialOrd`: Verlangt die Implementation einer Funktion um mit Instanzen des gleichen Typs sortiert werden zu können. Erlaubt aber auch, dass Werte zueinander nicht sortierbar sind. Dies ist zum Beispiel für Fließkommazahlen wichtig, da laut

IEE754 `Nan` nicht sortiert werden kann (weder `Nan <= 0` noch `Nan > 0` ergibt `true`) [**wiki:nan**][**rust:orly_programming**][**rust:doc:partialord**].

- `Ord`: Erzwingt im Gegensatz zu `PartialOrd`, dass Werte zueinander immer geordnet werden können.
- `Drop`: Verlangt die Implementation einer Funktion, die kurz vor der Speicherfreigabe eines Objekts aufgerufen wird (ähnlich Destruktor aus C++).

Mit dem Attribute `#[derive(..)]` ist eine automatisierte Implementation genannter Merkmale oft möglich, insofern die jeweiligen Bedingungen erfüllt sind. So kann im allgemeinen `#[derive(Clone)]` genutzt werden um eine Datenstruktur oder eine Aufzählung automatisch klonbar zu machen oder `#[derive(Debug)]` um automatisch alle Felder in Text wandeln zu können. Ein ergonomisches aber auch Fehler reduzierendes Feature.

2.10 Zugriffsmodifikatoren

Zugriffsmodifikatoren erlauben es in Rust, Module, Datenstrukturen, Aufzählungen, Merkmale und Funktionen gegenüber Nutzern einer Crate und anderen Modulen sichtbar zu machen. Der standardmäßige Zugriffsmodifikator limitiert die Sichtbarkeit auf das Modul, in dem die Deklaration stattgefunden hat und wird durch keine Notation eines Zugriffsmodifikators erreicht. Um die Sichtbarkeit auf die gesamte Crate zu erhöhen, wird ein `pub(crate)` vorangestellt. Mit `pub` ist die Deklaration für alle sichtbar.

Zugriffsmodifikatoren können auch vor `use` Anweisungen geschrieben werden, um entsprechende Datentypen zusätzlich unter einem neuen Namensraum bekannt zu machen.

2.11 Musterabgleich

Der `match` Ausdruck ist ein sehr mächtiges Werkzeug in Rust und entspricht einem stark erweiterten `switch` aus Programmiersprachen wie C, Java oder C#. Mit ihm ist es nicht nur möglich einen Wert einer Aufzählung aufzulösen sondern Muster inklusive Konstanten zu vergleichen und gleichzeitig auf eventuell beinhaltete Werte zuzugreifen oder diese zu konsumieren. In einem `match` wird immer der erste kompatible Codepfad ausgeführt.

```
1 fn main() {
2     let value : Option<&str> = Some("text");
3     match value {
4         Some("test") => println!("Nur ein Test"),
5         Some(value) => println!("Wert ist: {}", value),
6         None => println!("Kein Wert"),
7     };
```

```
8 }
```

Listing 2.8: Kompletter `match` Ausdruck

Die Ausgabe des Programms aus Listing 2.8 ist `Wert ist: text`. In dem Beispiel ist `value` aus Zeile 2 und 3 `Some("text")`. Sowohl Zeile 4 als auch Zeile 5 prüfen auf die Variation `Some`, aber nur der Codepfad in Zeile 5 wird ausgeführt. Dies liegt an der zusätzlichen Prüfung für den beinhalteten Wert, der für den Codepfad in Zeile 4 mit `"test"` übereinstimmen müsste. Da eine Übereinstimmung nicht vorliegt, trifft als nächstes Zeile 5 zu, in der nur die Variation `Some` übereinstimmen muss. Die Variable `value` bindet bei dieser Übereinstimmung den Wert, um ihn für den Programmcode ansprechbar zu machen. Falls dies nicht nötig wäre, könnte stattdessen auch die Wildcard `_` verwendet werden.

Das `match` Statement von Rust verlangt, dass eine Musterabgleichung immer zu einem Ergebnis führt. Dementsprechend müssen entweder alle Varianten eine Aufzählung aufgeführt sein oder ein Standardpfade vorhanden sein `_ => { }`. Hiermit wird verhindert, dass, nachdem eine Aufzählung um eine Variation erweitert wurde, eine Musterabgleichung nicht um das neue Element ergänzt wurde.

Wenn sogar nur ein konkreter Fall von Bedeutung ist, kann dies in der verkürzten `if let` Schreibweise notiert werden:

```
1 fn main() {
2     let mut value : Option<u32> = Some(4);
3     if let Some(ref mut value) = value {
4         *value += 1;
5     }
6     println!("{:?}", value); // "Some(5)"
7 }
```

Listing 2.9: Vereinfachte `if let` Ausdruck

Ein weiterer Unterschied von Listing 2.9 gegenüber Listing 2.8 ist in Zeile 3 das Schlüsselwort `ref`, wodurch der Konsum des Wertes verhindert wird. Das Schlüsselwort `mut` erlaubt zudem eine Änderung des Wertes, weswegen `value` in Zeile 4 vom Typ `&mut u32` ist. Die Dereferenzierung mit Addition wird somit ermöglicht.

Als Wildcard für sowohl nicht benötigte Werte, als auch alle weiteren Fälle kann `_` verwendet werden: `if let Some(_) = value { println!("It's something!"); }`

Weitere Möglichkeiten, Muster zu erkennen sind ab Seite 221 in [**rust:orly_programming**] in detaillierter Ausführung zu finden. Dazu gehören unter anderem die „guard expression“, „bindings“ und „ranges“. Aufgrund des Umfangs und die Irrelevanz für diese Arbeit wird hier auf eine weitere Vertiefung verzichtet.

2.12 Attribute

TODO: Unterscheidung Methode, Datentyp oder main.rs/lib.rs

2.13 Automatisierte Tests

TODO: arg1

2.14 Namens- und Formatierkonvention / Styleguide

```
enum MY_ENUM {
    AN_ENTRY,
    ANOTHER_ENTRY,
}
```

Listing 2.10: Beispiel für nicht Styleguide konformer Aufzählung

[warning]: **type ‘MY_ENUM‘ should have a camel case name such as ‘MyEnum‘**
[warning]: **variant ‘AN_ENTRY‘ should have a camel case name such as ‘AnEntry‘**
[warning]: **variant ‘ANOTHER_ENTRY‘ should have a camel case name such as ‘AnotherEntry‘** warning: unused variable: ‘a‘ [**rust:styleguide**]

TODO: official format/naming convetion, use, function, macro

TODO: type safety langauge

2.15 Niemals nichts und niemals unbehandelte Ausnahmen

Rust kennt `NULL` (-Pointer) nicht und erlaubt auch keine nicht initialisierte Variablen (siehe Unterabschnitt 2.22.1), bietet aber einen `Option<_>`-Datentyp als Ersatz an. Dieser Datentyp erzwingt eine Prüfung vor dem Zugriff auf den optionalen Wert (siehe Unterabschnitt 2.22.2).

Für die Fehlerbehandlung wird nicht auf ein Exception-Handling zurückgegriffen, sondern ein eigener Datentyp angeboten, der entweder den Rückgabewert enthält, oder aber einen Fehler: `Result<_, _>` (siehe Unterabschnitt 2.22.3).

Durch den Fragezeichenoperator kann trotzdem ein ähnliches Verhalten wie beim auftreten einer Ausnahme in Java oder C++ erzielt werden (siehe Unterabschnitt 2.22.3).

2.16 Besorgter Compiler

TODO: many warnings

TODO: remove?

2.17 Standardbibliothek

Das Rust Entwicklerteam ist darum bemüht, die Standardbibliothek sehr leichtgewichtig zu halten. Nicht eindeutig als fundamental eingestufte Funktionalitäten werden lieber als Crate auf `https://crates.io` angeboten, anstatt sie in die Standardbibliothek zu übernommen TODO: find example again. Mit dieser Entscheidung soll auch eine Entwicklung unabhängig von den Releasezyklen von Rust ermöglicht werden TODO: find source again.

Die Standardbibliothek ist selbst eine Crates, auf die standardmäßige eine Abhängigkeit erstellt wird. Für Fälle, in denen diese Abhängigkeit zu schwergewichtig ist, wie zum Beispiel im Embedded-Bereich, kann diese Abhängigkeit durch das Attribut `#![no_std]` unterbunden werden. Daraufhin sind nur noch die in der `core` Crate zur Verfügung gestellten, fundamentalen Sprachkonstrukte verwendbar.

In dieser Abschlussarbeit wird der volle Funktionsumfang der Standardbibliothek genutzt. Wichtige aber auch bekannte Datentypen sind hierbei:

- **std::vec::Vec**: Ein Vektor (wie eine Liste), bei dem die Werte in einem dynamisch groß allokierten Speicherbereich auf dem Heap liegen. Ist **der** Ersatz für dynamische Arrays, da auch der `[]`-Operator überschrieben ist und sich daher ein **Vec** wie ein Array ansprechen lässt.

  In Abbildung 2.1 ist das Speicherlayout eines **Vec** und einer **Slice** auf dem Stack und dem Heap abgebildet. Zu sehen ist, dass eine **Slice** direkt auf die Elemente eines **Vec** zeigen kann und sich daher von einem Array-Pointer aus C und C++ nur durch die angehängte Längeninformation unterscheidet.

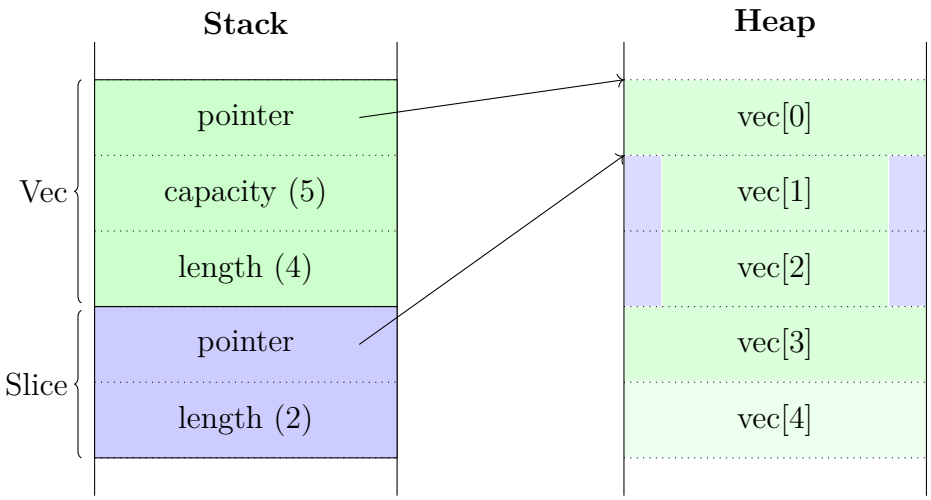

Abbildung 2.1: Speicherlayout Vec und Slice [**rust:orly_programming**]

- **std::boxed::Box**: Verweis auf einen Speicherbereich auf dem Heap für einen beliebigen Datentyp. Erlaubt es, u.a. Eigentümerschaft über ein unbekannt großen Datentyp zu erlangen, da dies die Größe einer **Box** nicht beeinflusst (siehe Abschnitt 2.9). Eine **Box** kann mit einem immer gültigen Heap-Pointer aus C und C++ verglichen werden.
- **std::string::String**: Eine UTF-8 codierte, vergrößer- und verkleinerbare Zeichenkette auf dem Heap.
- **std::rc::Rc**: Erweitert die **Box** um einen Referenzzähler und ermöglicht somit augenscheinlich mehrere Eigentümer, mit der Limitierung, nur noch lesend auf den beinhalteten Wert zugreifen zu können. Der beinhaltende Wert wird erst bei Lebensende der letzten **Rc** Instanz freigegeben. Verwendet einen mit wenig Mehraufwand verbundenen, nicht-atomaren Referenzzähler, weswegen eine **Rc** Instanz nicht zwischen Threads übertragen werden kann (`!Sync` , `!Send`).
- **std::sync::Arc**: Entspricht weitestgehend dem **Rc**, verwendet jedoch einen atomaren Referenzzähler. Dies ist zwar mit höheren Laufzeitkosten verbunden, erlaubt es aber, dass eine **Arc** Instanz zwischen Threads übertragen werden kann. Mehrere Threads können daher lesend auf den beinhalteten Wert zugreifen.
- **std::sync::Mutex**: TODO: ? Schützt Daten anstatt Code
- **std::sync::RwLock**: TODO: ? Erlaubt mehrfach lesend oder einmal schreibend
- **std::net::TcpStream**: TODO: ?
- **Module std::thread**: TODO: ?
- **std::collections::HashMap**: TODO: ?

2.18 Speichermanagement

Rust benutzt ein „statisches, automatisches Speicher Management – keinen Garbage Collector“ [**rust:youtube:goto2017**]. Das bedeutet, die Lebenszeit einer Variable wird statisch während der Compilezeit anhand des Geltungsbereichs ermittelt (siehe Abschnitt 2.18). Durch diese statische Analyses findet der Compiler heraus, wann der Speicher einer Variable wieder freigegeben werden muss. Dies ist genau dann, wenn der Geltungsbereich des Eigentümers zu Ende ist. Weder ein GC[5], der dies zur Laufzeit nachverfolgt, noch ein manuelles eingreifen durch den Entwickler (zum Beispiel durch `free(*void)`, wie in C/C++ üblich) ist nötig.

Falls der Compiler keine ordnungsgemäße Nutzung feststellen kann, wie zum Beispiel eine Referenz die länger als die eigentliche Variable lebt, wird die Kompilation verweigert. Der menschliche Faktor als Fehlerquelle wird wieder unterbunden, ohne Laufzeitkosten zu erzeugen (siehe Unterabschnitt 2.22.4).

Im folgenden Listing 2.11 wird Beispielhaft Speicher auf dem Heap allokiert. Dieser wird ordnungsgemäß freigegeben, ohne manuell eine Freigabe einzuleiten.

```
fn main() { // neuer Scope
    let mut a = Box::new(5); // 5 kommt auf den Heap
    { // neuer Scope
        let b = Box::new(10); // 10 kommt auch auf den Heap
        *a += *b; // a ist nun 15
    } // Lebenszeit von b zuende, Speicher wird freigeben
    println!("a: {}", a); // Ausgabe: "a: 15"
} // Lebenszeit von a zuende, Speicher wird freigegeben
```

Listing 2.11: Geltungsbereich von Variablen

Eine Variable kann auch vorzeitig durch den Aufruf von `std::mem::drop(_)` freigegeben werden. Die optionalen Implementation des `std::op::Drop`-Merkmals (siehe Abschnitt 2.9) kommt der Implementation des Destruktors aus C++ gleich.

2.19 Eigentümer- und Verleihprinzip

Bereits 2003 beschreibt Bruce Powel Douglass im Buch „Real-Time Design Patterns“, dass „passive“ Objekte ihre Arbeit nur in dem Thread-Kontext ihres „aktiven“ Eigentümers tätigen sollen [**douglass2003real**]. In dem beschriebenen „Concurrency Pattern“

[5] Garbage Collector

werden Objekte eindeutig Eigentümern zugeordnet, um so eine sicherere Nebenläufigkeit zu erlauben.

Diese Philosophie setzt Rust direkt in der Sprache um, denn in Rust darf ein Wert immer nur einen Eigentümer haben. Zusätzlich zu einem immer eindeutig identifizierbaren Eigentümer, kann der Wert auch ausgeliehen werden, um einen kurzzeitigen Zugriff zu erlauben; entweder exklusiv mit sowohl Lese- als auch Schreiberlaubnis, oder mehrfache mit nur Leseerlaubnis.

Eigentümerschaft kann auch übertragen werden, der vorherige Eigentümer kann danach nicht mehr auf den Wert zugreifen. Ein entsprechender Versuch wird mit einem Compilerfehler bemängelt.

Die Garantie, nur einen Eigentümer, eine exklusive Schreiberlaubnis oder mehrere Leseerlaubnisse auf eine Variable zu haben, wird durch die statische Lebenszeitanalyse garantiert (siehe Abschnitt 2.18). Da dies zur Compilezeit geschieht, ist eine Überprüfung zur Laufzeit nicht nötig, weshalb diese Philosophie keinen Laufzeitkosten mit sich bringt.

```
fn main() {
    let mut a = Box::new(1.0_f32); // Eigentümer der neuen
                                   // Heap-Variable ist a

    {
        let b = &a; // a wird an b mit Lesezugriff verliehen
        let c = &a; // a wird an c mit Lesezugriff verliehen

        println!("a: {}", a); // "a: 1"
        println!("b: {}", b); // "b: 1"
        println!("c: {}", c); // "c: 1"

        // let d = &mut a; // Nicht erlaubt: Es existieren
                           // verliehene Lesezugriffe

        // *a = 7_f32; // Nicht erlaubt: Es existieren
                       // verliehene Lesezugriffe

    } // Ende von b und c, a nicht mehr verliehen

    {
        let e = &mut a; // Leihe a mit Schreiberlaubnis
        **e = 9_f32;    // Setze Inhalt von a

        // println!("a: {}", a); // Nicht erlaubt: exklusiver
                                 // Zugriff an e verliehen

```

```
28          println!("e: {}", e); // "e: 9"
29
30      } // Ende von e, a nicht mehr verliehen
31
32      println!("a: {}", a); // "a: 9"
33      let f = a; // Neuer Eigentümer der Heap-Variable ist f
34      // *a = 12.5_f32; // Nicht erlaubt: Nicht mehr Eigentümer
35      // *f = 12.5_f32; // Nicht erlaubt: f nicht änderlich
36      println!("f: {}", f); // "f: 9"
37  }
```

Listing 2.12: Eigentümer und Referenzen von Variablen

2.20 Rust als funktionale Programmiersprache ??

TODO: functional programming -> no global state, no exceptions, find literature

TODO: prove via code

2.21 Rust als Objekt-Orientierte Programmiersprache ??

Vererbung explizit nicht erwünscht, Composition over Inheritance, inheritance dissallows static sizes, enum allow passing by value

TODO: trait

TODO: prove via design patterns, a few? from faq:: Is Rust object oriented? It is multi-paradigm. Many things you can do in OO languages you can do in Rust, but not everything, and not always using the same abstraction you're accustomed to.

2.22 Versprechen von Rust

> *„It's not bad programmers, it's that C is a hostile language“* [**rust:c_is_hostile_mena**]
>
> *„I'm thinking that C is actively hostile to writing and maintaining reliable code“* [**rust:c_is_hostile_mena**]

Rust wirbt mit Versprechen und Garantien, die dafür sorgen sollen, typische Fehler zu vermeiden. In einer perfekten Welt wären viele dieser Maßnahmen nicht nötig, da perfekte Wesen niemals einen Fehler machen und niemals etwas übersehen würden. Programmierer sind aber Menschen, Menschen machen Fehler. Deswegen hat Rust einige Interessante Mechaniken eingeführt, bekannte Fehlerquellen zu unterbinden und erzwingt die Einhaltung dieser, indem andere Vorgehensweisen meist ausgeschlossen werden.

Dieses Kapitel beschäftigt sich mit den wichtigsten und bekanntesten dieser Mechaniken.

2.22.1 Kein undefiniertes Verhalten

Bei der Entwicklung von Rust wird ein sehr großer Fokus darauf gelegt, keine undefinierten Zustände zu erlauben. Daher ist es normalerweise nicht möglich, ein undefiniertes Verhalten oder einen undefinierten Zustand zu erzeugen. Die Ausnahme bilden einige Fälle innerhalb von `unsafe` Blöcken, für zum Beispiel FFI (siehe Abschnitt 2.23). Für diese Fälle gibt es eine überschaubare Liste von Szenarien, aus denen ein undefinierter Zustand bzw. undefiniertes Verhalten resultieren kann [**rust:book:undefined**].

Als einfaches Beispiel eines undefinierten Zustandes in C ist eine Variable, die deklariert wurde, der aber noch keinen Wert zugewiesen wurde. In manchen Szenarien hat die Variable dann den Wert der in diesem Moment an der entsprechenden Stelle im Speicher steht, in anderen Szenarien wird der Speicher vom Betriebssystem, Allokator oder von vom Compiler eingefügten Befehlen mit 0en gefüllt – eine sichere Aussage ist nicht möglich. Sich darauf zu verlassen, dass neue Werte automatisch mit 0 initialisiert wurden, kann auf neuen Systemen oder mit anderen Compilern ein unvorhersehbares Verhalten provozieren.

Rust lässt deshalb keinen Zugriff auf Variablen zu, die nicht zuvor initialisiert wurden [**rust:orly_programming**]. Der Compiler stoppt mit einem Fehler: „**error[E0381]: use of possibly uninitialized variable: ‘a‘**“.

2.22.2 Keine vergessene Null-Pointer Prüfung

> *„I call it my billion-dollar mistake. It was the invention of the null reference in 1965“* [**rust:infoq:null**] TODO: cant find moment in video / presentation of this qutoe!?

Wie in Abschnitt 2.15 bereits erwähnt, kennt Rust keinen `NULL`-Pointer. Daher ist es auch nicht möglich, durch Nachlässigkeit auf den falschen Speicher zuzugreifen. Eine Referenz ist immer gültig. Für fälle, in denen es situationsbedingt keinen gültigen Wert gibt, bietet Rust stattdessen den `Option<_>` Datentyp an. `Option<_>` ist eine Aufzählung, die entweder `None` ohne einen Wert, oder `Some(_)` mit einem Wert ist. Auf den Wert kann nicht zugegriffen werden, ohne zu prüfen, ob wirklich die Variation `Some(_)` vorliegt. Dies kann

durch `match` oder verkürzt durch ein `if let Some(wert) = optional { /* tu etwas mit wert *` geschehen (siehe Abschnitt 2.11).

In vielen fällen kann der `Option<_>` Datentyp in Maschinencode als `NULL`-Pointer dargestellt werden, weswegen durch diese Abstraktion keine weiteren Laufzeitkosten eingeführt werden [**rust:orly_programming**] (siehe Unterabschnitt 2.22.6).

2.22.3 Keine vergessene Fehlerprüfung

```
1 #include <stdio.h>
2
3 void main(void) {
4     FILE *file = fopen("private.key", "w");
5     fputs("42", file);
6 }
```

Listing 2.13: Negativbeispiel: Fehlende Fehlerprüfung in C

In Listing 2.13 sind mindestens zwei Fehler versteckt, die aber keinen Compileabbruch auslösen, sondern sich zur Laufzeit zeigen können. Der erste Fehler ist eine fehlende Überprüfung des Rückgabewertes von `fopen` in Zeile 4. Der Rückgabewert kann `NULL` sein, falls das Öffnen der Datei fehlgeschlagen ist. Der Versuch in die Datei zu schreiben in Zeile 5 kann daraufhin in einen Speicherzugriffsfehler resultieren und das Programm abstürzen lassen.

In Rust wird weder eine Ausnahme geworfen, noch ein Rückgabewert zurück gegeben, der ohne Prüfung verwendet werden kann:

```
1  use std::fs::File;
2  use std::io::Write;
3
4  fn main() {
5    match File::create("private.key") {
6      Err(e) => println!("Datei nicht erstellbar: {}", e),
7      Ok(mut file) => {
8        if let Err(e) = write!(file, "42") {
9          println!("Konnte nicht in Datei schreiben: {}", e);
10       }
11     }
12   }
13 }
```

Listing 2.14: Positivbeispiel: Keine fehlende Fehlerprüfung in Rust

Der Rückgabewert von `File::open("private.key")` in Zeile 5 von Listing 2.14 ist vom Typ `Result<File, Error>`. Auf den eigentlichen Rückgabewert `File` kann nicht ohne eine Fehlerprüfung zugegriffen werden, da dies `Result` verhindert. Eine Fehlerprüfung kann wie in Zeile 5 mit einem `match` oder verkürzt durch ein `if let` wie in Zeile 8 geschehen.

Durch die statische Lebenszeitanalyse (siehe Abschnitt 2.18) in Rust ist der Geltungsbereich der `mut file` Variable bekannt, deshalb wird in dem Beispiel in Rust in Listing 2.14 die Datei auch wieder ordnungsgemäß geschlossen. Dies ist im im C Beispiel in Listing 2.13 nicht der Fall. In einem größeren Programm könnte so zu unbekanntem Zeitpunkt das Limit an gleichzeitig geöffneten Dateien erreicht werden.

Da eine `match` oder ein `if let` für jeden Funktionsaufruf der einen Fehler zurückgeben könnte, sehr umständlich und bereits für kleine Beispiele wie Listing 2.14 unübersichtlich wird, kann dies durch den Operator `?` abgekürzt werden. Dazu muss die Funktion, die den Operator verwendet aber auch ein `Result` im einem kompatiblen Fehlertyp zurückgeben, wie in Listing 2.15 zu sehen:

```rust
use std::fs::File;
use std::io::Write;
use std::io::Error;

fn main() {
  if let Err(e) = schreibe_schluessel("private.key", "42") {
    println!("Fehler aufgetreten: {}", e);
  }
}

fn schreibe_schluessel(file: &str, content: &str) ->
   Result<(), Error>  {
  let mut file = File::create(file)?;
  write!(file, "{}", content)?;
  Ok(())
}
```

Listing 2.15: Verkürzte Fehlerbehandlung in Rust

2.22.4 No dangling pointer

TODO: src https://www.youtube.com/watch?v=d1uraoHM8Gg

2.22.5 Sichere Nebenläufigkeit

TODO: „Safety is invisible“ [**rust:orly_programming**]

TODO: Send, Sync, No dataraces weil Ownership Abschnitt 2.19, Channel, Mutex, RwLock

TODO: Datarace benötigt immer einen schreibenden + min einen lesenden gleichzeitig

TODO: Mutex, RwLock – immer mit Result

2.22.6 Zero Cost Abstraction

Trotz der vielen verwendeten Abstraktionen möchte Rust dadurch möglichst keine weitere Laufzeitkosten erzeugen. Beim übersetzen werden deshalb viele Abstraktionen durch Optimierungen für den Maschinencode unsichtbar.

Der `Option<_>` Datentyp kann zum Beispiel in vielen Fällen als Pointer dargestellt werden, der bei `NULL` `None` und ansonsten `Some(_)` ist [**rust:orly_programming**]. Somit wird eine Überprüfung erzwungen, ohne dabei Laufzeitkosten erzeugt zu haben.

Ein weiteres Beispiel sind die Referenzzählertypen `Rc` und `Arc<_>`. Der Zähler ist im Heap direkt vor dem beinhalteten Wert und nicht in einem extra Speicherbereich, weshalb ein weiterer, indirekter Speicherzugriff mit Laufzeitkosten verhindert werden kann.

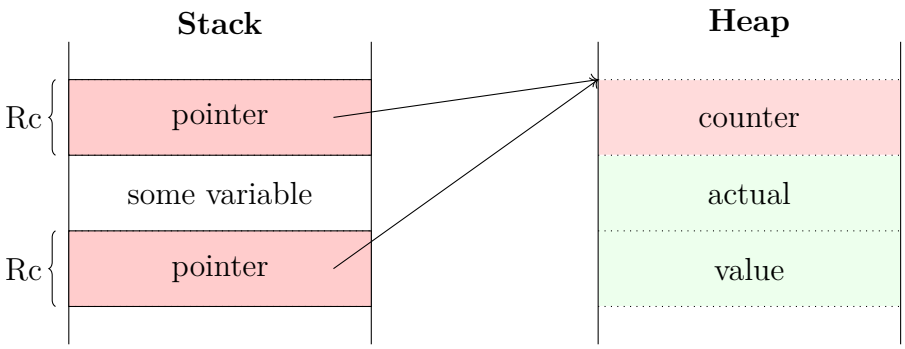

Abbildung 2.2: Speicherlayout Rc [**rust:orly_programming**]

2.23 Einbinden von externen Bibliotheken

Externe Datentypen

Rust bietet durch das Foreign Function Interface[6] die Möglichkeit, andere (System-)Bibliotheken einzubinden. Entsprechende Strukturen und Funktionen werden durch einen `extern` Block oder im Falle von Strukturen stattdessen optional mit einem `#[repr(C)]` gekennzeichnet.

In einem Beispiel, soll die Nutzung von Foreign Function Interface demonstriert werden.

[6] Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [**wiki:ffi**]

```c
typedef struct PositionOffset {
    long position_north;
    long position_east;
    long *std_dev_position_north; // OPTIONAL
    long *std_dev_position_east;  // OPTIONAL

    // ...
} PositionOffset_t;
```

Listing 2.16: Ausschnitt von „PositionOffset“ (C-Code) aus der *libmessages-sys* Crate

Die Struktur in Listing 2.16 muss zur Nutzung in Rust zuerst bekannt gemacht werden. Dabei gibt es mehrere Möglichkeiten:

- Falls der Aufbau der Struktur nicht von Bedeutung ist, kann es ausreichen, den Datentyp lediglich bekannt zu machen: `#[repr(C)] struct PositionOffset;`. In diesem Fall können aber nur Referenzen und Raw-Pointer auf die Struktur verwendet werden.
- Falls der Aufbau wie in 2.23 unbedeutend ist, es soll aber ausdrücklich auf einen externen Datentyp hingewiesen werden, kann dieser in einem `extern { }` Block bekannt gemacht werden: `extern { type PositionOffset; }` [**rust:github:extern_type**]. Dies ist zum jetzigen Zeitpunkt aber nur in „nightly“ und hinter dem „feature gate“ `extern_types` möglich.
- Der Inhalt der Struktur ist von Bedeutung, da darauf zugegriffen oder in Rust eine Instanz werden soll. In diesem Fall ist eine komplette Wiedergabe die Struktur unumgänglich:

```rust
use std::os::raw::c_long;

#[repr(C)]
pub struct PositionOffset {
    pub position_north: c_long,
    pub position_east: c_long,
    pub std_dev_position_north: *mut c_long,
    pub std_dev_position_east: *mut c_long,
    // ...
}
```

Listing 2.17: Ausschnitt von „PositionOffset“ (Rust-Code) aus der *libmessages-sys* Crate

In Listing 2.17 ist die Struktur „PositionOffset“ deklariert, die durch das Attribut `#[repr(C)]` wie eine C-Struktur im Speicher organisiert wird. Damit die Struktur in Rust kompatibel zu der in C ist, müssen die Variablen von der selben Größe sein, ansonsten würde das Speicherlayout nicht übereinstimmen. Hierfür werden spezielle Datentypen (`c_long` , `c_void` , `c_char` , ...) angeboten, um die Kompatibilität mit verschiedenen Systemen und C-Compilern zu wahren.

Ein C-Pointer `*long` wird in Rust „Raw-Pointer“ genannt und entweder `*mut c_long` oder `*const c_long` geschrieben. Der Unterschied ist wie zwischen `&mut c_long` und `&c_long` und dient dem Rust Compiler zum Nachvollziehen, ob ein exklusiver zugriff benötigt wird, oder nicht. Dies hilft zwar für die Fehlervermeidung durch eventuelle Compilefehler anstatt Laufzeitfehler, ist aber für die C-Funktion unbedeutend [**rust:book:raw_ptr**]:

Referenz in Rust	Raw-Pointer in Rust	C-Pointer
`&mut c_long`	`*mut c_long`	`long*`
`&c_long`	`*const c_long`	`long*`

Abbildung 2.3: Vergleich Rust Raw-Pointer und Referenz zu C-Pointer

Externer Funktionsaufruf

Externe Funktionen müssen im Gegensatz zu externen Strukturen immer in einem `extern {}` Block deklariert sein.

```
1  use std::os::raw::c_void;
2
3  #[link(name = "messages", kind = "static")]
4  extern {
5      type asn_TYPE_descriptor_s;
6      type asn_enc_rval_t;
7
8      fn uper_encode_to_buffer(
9          type_descriptor: *const asn_TYPE_descriptor_s,
10         struct_ptr: *const c_void,
11         buffer: *mut c_void,
12         buffer_size: usize,
13     ) -> asn_enc_rval_t;
14 }
```

Listing 2.18: Externe Funktionsdefinition der ASN.1 Funktion zum Enkodieren

Wie in Listing 2.18 zu sehen ist, können auch `extern {}` Blöcke mit Attributen versehen werden. Zwingend ist bei der Verwendung eines `#[link(..)]` Attributes der Name der Bibliothek, auf die sich der im `extern {}` Block stehende Code bezieht. Optional kann auch wie in Listing 2.18 die Art der Linkung (dynamisch oder statisch) angegeben werden.

Die Art der Definition einer externen Funktion unterscheidet sich nicht von einer normalen Funktionsdefinition. Es sollten aber, wie in Abschnitt 2.23 beschrieben, zu C bzw. der externen Sprache kompatiblen Datentypen verwendet werden.

2.24 Kernfeatures

TODO: nothing on heap unless specified (Box, Vec, other container)

TODO: closures are fast, orly, p.310

https://www.youtube.com/watch?v=d1uraoHM8Gg
TODO: no need for a runtime, all static analytics
TODO: memory safety
TODO: data-race freedom
TODO: active community
TODO: concurrency: no undefined behavior
TODO: ffi binding Foreign Function Interface
TODO: zero cost abstraction
TODO: package manager: cargo

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: static type system with local type inference
TODO: explicit notion of mutability
TODO: zero-cost abstraction *(do not introduce new cost through implementation of abstraction)
TODO: errors are values not exceptions TODO: no null
TODO: ßtatic automatic memory management no garbage collection
TODO: often compared to GO and D (44min)

2.25 Schwächen

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: compile-times
TODO: Rust is a vampire language, it does not reflect at all!
TODO: depending on the field -> majority of libraries?

2.26 Performance Fallstricke

TODO: [**rust:performance_pitfalls**]

2.27 Beispiele von Verwendung von Rust

TODO: firefox

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: GTK binding heavily to rust

TODO: unstable TODO: ffi

3 Hochperformante, serverbasierte Kommunikationsplattform

Dieses Kapitel erläutert den Begriff „hochperformante, serverbasierte Kommunikationsplattform“ und vermittelt Basiswissen hierzu.

3.1 Echtzeitsysteme

Echtzeitsysteme zeichnen sich im allgemeinen dadurch aus, eine Aufgabe in einem zuvor vorgegebenen Zeitraum bearbeiten zu können. Es existiert zu einer Aufgabe also immer eine Frist. Bei der Bewertung der Richtigkeit eines Systems, wird die Fähigkeit, eine Frist einhalten zu können, auch bewertet [**perf:buttazzo2006soft**]. Je nach Art des Echtzeitsystems, wird diese First jedoch unterschiedlich gewichtet:

- Bei einem harten Echtzeitsystem kann eine Überschreitung der Frist einen katastrophalen Ausgang haben. Selbst im schlimmsten Fall darf diese Frist nicht überschritten werden. Deswegen wird in einem harten Echtzeitsystem die im maximale Reaktionszeit dem Zeitraum bis zur Frist gegenübergestellt [**douglass2003real**]. Ein Ergebnis nach Ablauf der Frist wird als nutzlos gewertet [**perf:wang2017real**].

  Zum Beispiel könnte eine zu späte Auswertung von Beschleunigungsdaten in einem Flugzeug zu einer verzögerten und mittlerweile falschen Reaktion und daraufhin zu einem Absturz führen [**perf:laplante2004real**].

- Bei einem weichen Echtzeitsystem resultiert die Überschreitung des vorgegebenes Zeitraums nicht in eine Katastrophe. Es wird die durchschnittliche Reaktionszeit dem Zeitraum bis zur Frist gegenübergestellt, eine seltene und unter last auftretende Überschreitung wird in kauf genommen [**douglass2003real**]. Das System führt in so einem Fall weiterhin seine Aufgaben aus, die Performance wird aber TODO: abgewertet eingestuft. Weiche Echtzeitsysteme können sogar überhaupt keine Frist haben, sondern die Aufgabe, die Antwortzeit so gering wie möglich zu gehalten [**perf:buttazzo2006soft**].

3.1.1 Echtzeitnah

TODO: ?

3.2 Funktionale Sicherheit

3.2.1 Was ist dann ein hochperformantes System

3.2.2 Low-Latency + Entwurfsmuster + Patterns? + Algorithmen?

TODO: Hochperformant -> parallel?

TODO: Design Pattern, Gamma et al, four important aspects

TODO: Real Time Design Patterns Buch: Ab Seite 141, verschiedene Systempatterns, microkernel [**douglass2003real**]? channel architektur pattern [**douglass2003real**]?

TODO: Message Queuing Pattern [**douglass2003real**]

TODO: Clean Architecture / Clean Code

3.3 Serverbasierte Kommunikationsplattform: MEC

3.4 ASN.1

Die Notationsform ASN.1[1] ermöglicht abstrakte Datentypen und Werte zu beschreiben [**asn:layman**]. Die Beschreibungen können anschließend zu Quellcode einer theoretisch[2] beliebigen Programmiersprache compiliert werden. Beschriebene Datentypen werden dadurch als native Konstrukte dargestellt und können mittels einer der standardisierten (oder auch eigenen [**asn:itu:ecn**]) Encodierungen serialisiert werden.

Um den Austausch zwischen verschiedenen Anwendungen und Systemen zu ermöglichen, sind von der TODO: ITU bereits einige Encodierungen standardisiert [**asn:itu:x691**]. Für diese Arbeit ist aber einzig der PER Standard relevant, da der Server diese Encodierung verwenden muss, um mit den Sensoren und den Autos zu kommunizieren (siehe TODO: ref requirements / analyse).

Die anderen bekannteren Verfahren werden deshalb nur kurz erwähnt:

- **BER** (Basic Encoding Rules): Flexible binäre Encodierung [**asn:wiki:x690**], spezifiziert in X.690 [**asn:itu:x690**]
- **CER** (Canonical Encoding Rules): Reduziert BER mit der Restriktion die Enden von Datenfelder speziell zu Markieren anstatt deren Größe zu übermitteln, eignet sich gut für große Nachrichten [**asn:wiki:x690**], spezifiziert in X.690 [**asn:itu:x690**]

[1] Abstract Syntax Notation One

[2] Es gibt keine Einschränkungen seitens des Standards, aber entsprechende Compiler zu finden erweist sich als schwierig TODO: ref impl Schwierigkeiten mit ASN+Rust

- **DER** (Distinguished Encoding Rules): Reduziert BER durch die Restriktion Größeninformationen zu Datenfeldern in den Metadaten zu übermitteln, eignet sich gut für kleine Nachrichten [**asn:wiki:x690**], spezifiziert in X.690 [**asn:itu:x690**]
- **XER** (XML Encoding Rules): Beschreibt den Wechsel der Darstellung zwischen ASN.1 und XML, spezifiziert in X.693 [**asn:itu:x693**]

TODO: isdn

[**asn:itu:asn.1**]

> *„ ASN.1 has a long record of accomplishment, having been in use since 1984. It has evolved over time to meet industry needs, such as PER support for the bandwidth-constrained wireless industry and XML support for easy use of common Web browsers. “* [**asn:itu:asn.1**]

PER

Die Packed Encoding Rules werden in in X.691 [**asn:itu:x691**] beschrieben. Sie beschreiben eine Encodierung, die genutzt werden kann, um beschriebene Datentypen möglichst kompakt – also in wenigen Bytes – zu serialisieren.

TODO: sources: Für den Einsatz im Mobilfunknetz ist diese Encodierung sehr beliebt, da bei der Übermittlung einer Nachricht kein anderen Kommunikationsteilnehmer auf dieser Frequenz eine weiter Nachricht übermitteln kann. Eine kürzere Nachricht blockiert eine Frequenz kürzer, weshalb kürzere Nachrichten einen höheren Durchsatz erlaubt. Im Mobilfunkbereich ist dies von besonderer Bedeutung, da das Medium von vielen Teilnehmern gleichzeitig geteilt wird. TODO: michael.refactor_this_shit()

3.5 Sensordaten?

3.6 TCP?

4 Anforderungen

TODO: irrelevant? Safety / Funktionale Sicherheit Da bei Fehlern möglicherweise andere Verkehrsteilnehmer zu Schaden kommen können, müssen diverse Sicherheitsrichtlinien beachtet werden. Die Industrienorm ISO 26262 beschreibt dabei verschiedene Vorgehensweisen, unter anderem eine FBA[1], Risikoabschätzung durch Einstufung nach ASILs[2] und beschreibt Gegenmaßnahmen.

TODO: asn

TODO: mobile edge computer -> ubuntu linux

4.1 Funktionale Anforderungen

4.1.1 Anforderung 1: Implementation in Rust

Die Implementation wird in der Programmiersprache Rust vorgenommen.

4.1.2 Anforderung 2: Plattform MEC

Die Implementation des Servers muss auf einem MEC Server mit dem Betriebssystem TODO: Ubuntu 14.04 LTS Server ausführbar sein.

4.1.3 Anforderung 3: Reaktionszeit für Ergebnisse des Fusions-Algorithmus

Die Zeit die der Server für die Weitergabe der Ergebnisse aus dem Fusions-Algorithmus benögtit, soll TODO: trölf Millisekunden nicht überschreiten.

4.1.4 Anforderung 4: Kein Echtzeitsystem

Trotz Anforderung 3 wird das System nicht als Echtzeitsystem gewertet. Eine Analyse für die maximale Reaktionszeit ist nicht verlangt.

[1] Fehlerbaumanalyse
[2] Automotive Safety Integrity Levels

4.1.5 Anforderung 5: TCP Server

Auf Port TODO: ... werden auf neue TCP Verbindungen gehört. Jeder Client hat eine eigene TCP Verbindung.

4.1.6 Anforderung 6: Kommunikationsprotokoll ist ASN.1

Das Protokoll für die Kommunikation zwischen dem Server und den Clients ist ASN.1. Es werden die bereits definierten Nachrichten verwendet und keine neuen Nachrichten definiert.

4.1.7 Anforderung 7: Client als Sensor

Ein Client kann sich nach dem Verbindungsaufbau als Sensor registrieren.

4.1.8 Anforderung 8: Client als Fahrzeug

Ein Client kann sich nach dem Verbindungsaufbau als Fahrzeug registrieren.

4.1.9 Anforderung 9: GeoFence bestimmbar

Ein Client kann das GeoFence in dem er sich physikalisch befindet bekannt zuweisen.

4.1.10 Anforderung 10: GeoFence Unterteilung

Es wird zwischen aktiven und inaktiven GeoFences unterschieden. Ein GeoFence ist nur dann aktiv, wenn mindestens ein Fahrzeug zugewiesen ist.

4.1.11 Anforderung 11: Sensoren pausieren

Sensoren werden bei der Zustandsänderung des zugewiesenen GeoFences zu inaktiv oder bei Zuweisung zu einem inaktiven GeoFence pausiert.

4.1.12 Anforderung 12: Sensoren wecken

Sensoren werden bei der Zustandsänderung des zugewiesenen Geofences zu aktiv oder bei Zuweisung zu einem aktiven GeoFence geweckt.

4.1.13 Anforderung 13: Sensordaten weitergeben

Empfangene Sensordaten werden dekodiert und an den Fusions-Algorithmus weitergegeben. TODO: geofence?

4.1.14 Anforderung 14: Ergebnisse weitergeben

Ergebnisse des Fusions-Algorithmus werden enkodiert un an die Fahrzeuge in den entsprechenden GeoFences versendet.

4.1.15 Anforderung 15: Mindestanzahl Clients

Der Server muss mindestens TODO: .. Sensoren und TODO: .. Fahrzeuge gleichzeitig bedienen können.

4.1.16 Anforderung 16: Reaktionszeit für Sensordaten

Die Zeit die der Server für Anforderung 13 und 14 zusammen benötigt soll TODO: trölf Millisekunden nicht überschreiten.

4.1.17 Anforderung 17: Widerstand gegen Sensor DOS

Die Funktionalität des Servers gegenüber anderen Clients wird durch eine Überflutung von Daten eines Sensors nicht beeinträchtigt. TODO: optional?

4.1.18 Anforderung 18: Widerstand gegen Nachrichtenrückstau

Die Funktionalität des Servers gegenüber anderen Clients wird durch Fahrzeuge, für die sich ein TODO: Nachrichtenrückstau bildet und von einzelnen langsamen Verbindungen nicht beeinträchtigt. TODO: optional?

4.2 Nichtfunktionale Anforderungen

4.2.1 Anforderung 19: Möglichst schnell

Der Server soll auf Sensordaten und Algorithmusergebnisse schnell reagieren.

5 Systemanalyse

5.1 Systemkontextdiagramm

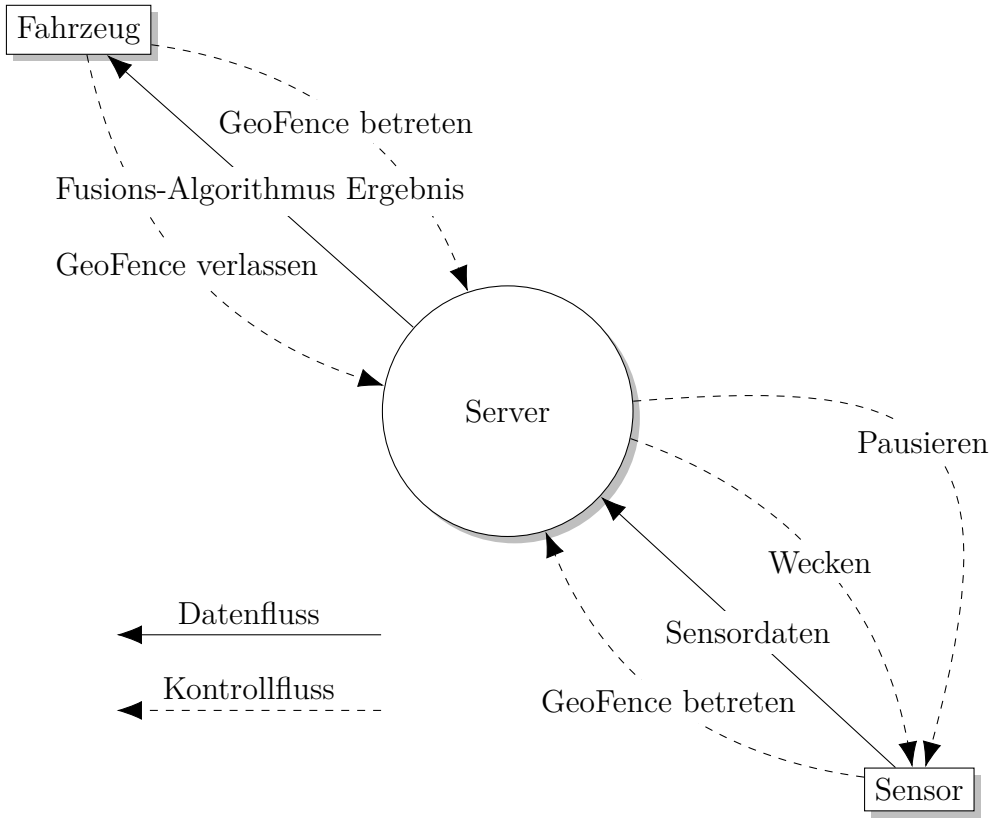

Abbildung 5.1: Systemkontextdiagramm

5.2 Komponentendiagramm oder sowas?

5.3 Use-Case Diagramme

TODO: was wirklich umgesetzt sein wird

5.4 Schnitstellenanalyse

6 Systementwurf

6.1 Architektur

6.2 Änderungen bedingt durch Rust

7 Implementierung

7.0.1 Unerwartete Schwierigkeiten

TODO: Schwierigkeiten: FFI binding, manuell -> meh, also generieren

8 Auswertung

9 Zusammenfassung und Fazit

Abkürzungsverzeichnis

ASIL Automotive Safety Integrity Level. 39

ASN.1 Abstract Syntax Notation One. 37

BMWi Bundesministerium für Wirtschaft und Energie. 3

FBA Fehlerbaumanalyse. 39

GC Garbage Collector. 23

MEC Mobile Edge Computing. 3, 4

Abbildungsverzeichnis

Listings