Evaluation der Programmiersprache Rust für den Entwurf und die Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens

Michael Watzko*, **Manfred Dausmann**, **NN**

Fakultät Informationstechnik der Hochschule Esslingen – University of Applied Sciences

Sommersemester 2018

Abbildung 1: Rust

Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Autos von Tesla 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 verschiedenen 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, 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.

*Diese Arbeit wurde durchgeführt bei der Firma IT-Designers GmbH, Esslingen

[1] Jim Blandy und Jason Orendorff: Programming Rust, 2017, ISBN: 1491927283

[2] Florian Gilcher: GOTO 2017 – Why is Rust Successful? https://www.youtube.com/watch?v=-Tj8Q12DaEQ

[3] B.P. Douglass. Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems, ISBN 9780201699562, 2003

Bildquellen: https://www.rust-lang.org/logos/rust-logo-256x256.png

Die Programmiersprache Rust

Rust hat als Ziel, eine sichere und performante Systemprogrammiersprache zu sein. Abstraktionen sollen die Sicherheit, Lesbarkeit und Nutzbarkeit verbessern aber keine unnötigen Performance-Einbußen verursachen.

Aus anderen Programmiersprachen bekannte Fehlerquellen – wie vergessene NULL-Pointer Prüfung, vergessene Fehlerprüfung, „dangling pointers“ oder „memory leaks“ – werden durch strikte Regeln und mit Hilfe des Compilers verhindert. Im Gegensatz zu Programmiersprachen, die dies mit Hilfe ihrer Laufzeitumgebung[1] sicherstellen, werden diese Regeln in Rust durch eine statische Lebenszeitanalyse und mit dem Eigentümerprinzip bei der Kompilation überprüft und erzwungen. Dadurch erreicht Rust eine zur Laufzeit hohe Ausführgeschwindigkeit.

Das Eigentümerprinzip und die Markierung von Datentypen durch Merkmale vereinfacht es zudem, nebenläufige und sichere Programme zu schreiben.

Speicherverwaltung

Rust benutzt ein „statisches, automatisches Speichermanagement – keinen Garbage Collector“ [2]. Das bedeutet, die Lebenszeit einer Variable wird statisch während der Compilezeit anhand des Geltungsbereichs ermittelt. 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 Garbage-Collector, 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 ihren referenzierten Wert überleben möchte, wird die Kompilation verweigert. Dadurch wird das Problem des „dangling pointers“ verhindert, ohne Laufzeitkosten zu erzeugen.

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 (Seite 204, [3]). In dem beschriebenen „Concurrency Pattern“ 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 einer Fehlermeldung durch den Compiler bemängelt.

Die Garantie, nur einen Eigentümer, eine exklusive Schreiberlaubnis oder mehrere Leseerlaubnisse auf eine Variable zu haben, wird durch die statische Lebenszeitanalyse garantiert.

Sichere Nebenläufigkeit

Eine sichere Nebenläufigkeit wird in Rust durch das Eigentümerprinzip in Kombination mit zusätzlichen Typmerkmalen erreicht. Dabei ist diese sichere Nebenläufigkeit meist unsichtbar (Seite 41, [1]), da der Compiler eine unsichere und damit syntaktisch falsche Verwendung nicht übersetzt. Ein Rust Programm das compiliert, ist daher, in vielerlei Hinsicht, sicher in der Nebenläufigkeit. Einzig ein „Deadlock“ kann nicht statisch ermittelt und verhindert werden.

Eine Wettlaufsituation (englisch „race condition“) um einen Wert ist in Rust nicht möglich. Das Eigentümer- und Leihprinzip verhindert dies, denn es kann nur exklusiv schreibend auf einen Wert zugegriffen werden. Für einen Datenwettlauf muss dagegen, gleichzeitig zu einem schreibenden, ein lesender Zugriff erfolgen.

Datentypen, die einen gemeinsamen Zugriff auf veränderliche Werte ermöglichen, liefern immer ein Ergebnis ob der Versuch, einen exklusiven Schreib- oder Lesezugriff zu erhalten, geklappt hat. Erst nach einer Fehlerauswertung kann auf den Wert zugegriffen werden.

Ausblick

Durch die u.a. hier erwähnten Garantien von Rust, soll eine funktional sichere Implementation einer Kommunikationsplattform, im Rahmen des MEC-View Forschungsprojektes, geschaffen werden. Im Umfeld des automatisierten Fahrens hat dies einen besonderen Stellenwert.

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