Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Invasives Rust

Bachelorarbeit von

Hermann Krumrey

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 1. Januar 1990 – 31. Dezember 2000

Zusammenfassung

Die Parallelisierung von Rechnern liegt immer mehr im Fokus der derzeitigen technologischen Entwicklung. Dies erfordert die Entwicklung und Nutzung von anderen Programmierparadigmen, welche effektiv diese neuen Architekturen ausnutzen können. Ein möglicher Ansatz ist hierbei das invasive Computing. Dieses ermöglicht es dem Programmierer, die Ressourcennutzung eines Programms feiner zu kontrollieren.

Das IRTSS Betriebssystem, welches eine beispielhafte Implementierung für ein solches invasives System bietet, unterstützt derzeit die Verwendung der Programmiersprachen C, C++ als auch X10. Hiermit wird die Einführung der Programmiersprache Rust als weitere unterstützte Sprache vorgeschlagen. Diese Sprache hat einige wünschenswerte Merkmale, welche interessant für den Gebrauch im invasiven Computing sind.

Inhaltsverzeichnis

1 Einführung

Die Parallelisierung von Rechnern liegt immer mehr im Fokus der derzeitigen technologischen Entwicklung. Dies erfordert die Entwicklung und Nutzung von anderen Programmierparadigmen, welche effektiv diese neuen Architekturen ausnutzen können. Ein möglicher Ansatz ist hierbei das invasive Computing. Dieses ermöglicht es dem Programmierer, die Ressourcennutzung eines Programms feiner zu kontrollieren.

Das IRTSS Betriebssystem, welches eine beispielhafte Implementierung für ein solches invasives System bietet, unterstützt derzeit die Verwendung der Programmiersprachen C, C++ als auch X10. Hiermit wird die Einführung der Programmiersprache Rust als weitere unterstützte Sprache vorgeschlagen. Diese Sprache hat einige wünschenswerte Merkmale, welche interessant für den Gebrauch im invasiven Computing sind.

2 Grundlagen und Verwandte Arbeiten

Im folgenden werden die Grundlagen der Programmiersprache Rust, des invasiven Computing und der Rechnerarchitektur SPARC behandelt.

2.1 Rust

Rust ist eine Programmiersprache, welche von Mozilla Research entwickelt wird [**?**]rustWikDe). Die Entwicklung der Sprache begann als persönliches Projekt des Mozilla-Mitarbeiters Graydon Hoare und wird seit 2009 von Mozilla unterstützt [**?**]rustWikDe). 2010 erschien die erste öffentliche Version der Sprache, im Jahr 2015 wurde die erste stabile Version veröffentlicht [**?**]rustWikDe). Sie erfreut sich in der jüngsten Vergangenheit wachsende Beliebtheit bei Programmierern aller Art. So wurde bei einer Umfrage der Webseite stackoverflow.com im Jahre 2016 als beliebteste Programmiersprache bei Entwicklern ermittelt. [1]

Das wohl derzeit prominenteste Projekt, welche Rust verwendet ist der Servo Layout-Engine, welcher gemeinsam von Mozilla und Samsung[2] entwickelt wird. Dieser Layout-Engine soll schrittweise den bisherigen Engine ‘Gecko‘, welcher in C++ geschrieben wurde, im Mozilla Firefox Webbrowser ersetzen und dabei deutlich bessere Leistung vorweisen.

2.1.1 Motivation

Eine der Kernziele der Rust Programmiersprache ist es, sichere Speicherzugriffe zu gewährleisten. Dies wird mithilfe des sogenannten “Ownership”-modells erreicht. Dieses Modell ermöglicht es, fehlerhafte Speicherzugriffe zu vermeiden, welche ein ernsthaftes Sicherheitsrisiko darstellen, denn diese können zu undefiniertem Verhalten führen, welches wiederum auch gezielt als Angriffsvektor genutzt werden kann. Diese Speichersicherheit wird erreicht, ohne dabei von einem Garbage Collector Gebrauch

zu machen. Dies macht Rust zu einer interessanten Alternative zu Systemsprachen wie C, welche nicht sicher bezüglich der Speicherverwaltung sind, als auch zu Sprachen mit Garbage Collector wie X10, welche durch den Garbage Collector in gewissen Situationen und Anwendungsbereichen, beispielsweise bei Echtzeitsystemen, nicht ideal sind.

Eine weitere Angriffsfläche als auch Fehlerquelle in herkömmlichen Systemprogrammiersprachen wie C und C++ ist das unsichere Typsystem. In Rust wird dies vermieden, indem ein typsicheres Typsystem verwendet wird.

Ein weiteres Ziel der Sprache ist die Geschwindigkeit. Rust soll vergleichbare Geschwindigkeiten zu anderen Systemsprachen wie C oder C++ erreichen. Dies kann in Benchmarks zumeist auch bestätigt werden[3]. Benchmarks, welche Rust mit höheren Programmiersprachen wie Java[4] oder Python[5] vergleichen, weisen oft einen signfikanten Vorspruch für Rust auf.

Das effiziente Einbinden mit anderen Sprachen ist eine zusätzliche Motivation hinter der Programmiersprache. So kann Rust beispielsweise mithilfe der sogenannten "Foreign Function Interface" Funktionen welche in anderen Programmiersprachen geschrieben sind benutzen. Außerdem ist es möglich, Rust Code aus anderen Programmiersprachen, inklusive höheren Programmiersprachen wie Python oder Ruby, aufzurufen.

Nebenläufigkeit ist ein weiteres Ziel der Sprache, mithilfe des Architektur der Sprache soll es zwischen einzelnen Threads nicht zu kritischen Wettläufen kommen, welches eines der Hauptfehlerquellen bei parallelem Programmieren beseitigt.

Rust soll dem Programmierer zudem höhere Abstraktionen bieten, welche nicht oder nur gering die Effizienz des Programm beinflussen. Diese sogenannten "Zero-cost Abstractions" erleichtern es Programmierern, welche nur wenig Erfahrung mit Systemsprachen vorweisen können, die Programmiersprache zu verwenden, ohne dabei auf Leistung verzichten zu müssen..

2.1.2 Grundlegende Eigenschaften der Sprache

Im folgenden werden die grundlegenden Eigenschaften der Programmiersprache Rust erläutert.

Das Ownership-Modell

Das signifikanteste Alleinstellungsmerkmal der Programmiersprache Rust ist das Ownership Modell. Erst hierdurch wird die Speichersicherheit ohne Garbage Collector möglich.

Die Grundlage des Ownership-Modells ist die Bindung von Speicher an eine Variable. Sobald eine Variable initialisiert wird, wird Speicher für diese alloziert. Sobald diese Variable sich allerdings nicht mehr im gültigen Anwendungsbereich befindet, wird dieser Speicher automatisch wieder freigegeben. Dies wird im folgenden Beispiel veranschaulicht:

```
fn foo() {
    let v = vec![1, 2, 3]; // Alloziert Speicher
}

fn main() {
    foo();
    // Speicher ist hier wieder freigegeben
}
```

Außerdem erlaubt es Rust keinen zwei Variablen auf denselben Speicher zu zeigen. So wird beispielsweise bei einer Zuweisung einer Variable zu einer anderen der “Besitz” des Speichers auf die neue Variable übertragen und die alte Variable kann nun nicht mehr verwendet werden. Dies geschieht auch wenn die Variable als Parameter in einem Funktionsaufrauf verwendet wird. Dieses Verhalten ist als “Move”-Semantik bekannt. Eine Veranschaulichung dessen folgt:

```
fn foo(v: Vec<i32>) {
    // Details unwichtig
}

fn main() {

    let v = vec![1, 2, 3];
    let v2 = v;  // v ist ab jetzt nicht mehr benutzbar!
    foo(v2);     // v2 ist ab jetzt nicht mehr benutzbar!

}
```

Durch dieses Verhalten wird sichergestellt, dass eine Variable exklusiven Zugriff auf die allozierten Speicherbereiche besitzt, wodurch fehlerhafte Speicherzugriffe vermieden werden.

Eine Ausnahme hierzu bilden Type, welche das “Copy-Trait” implementieren. So können beispielsweise variablen vom Typ i32, welches das “Copy-Trait” implementiert, beliebig oft wiederverwendet werden. In diesem Fall existieren trotzdem nicht mehrere Zeiger auf dieselbe Speicherregion, der Speicherinhalt wird stattdessen jedesmal kopiert.

Ein weiteres interessantes “Trait” welches ein Typ in Rust implementieren kann ist das “Drop-Trait”. Imlementiert ein Typ dieses, so kann zusätzlicher Code ausgeführt werden, sobald eine Variable von dem Typ den gültigen Anwendungsbereich verlässt. Dies kann hilfreich sein, um andere Speicherbereiche, welche in einem nicht-trivialen Zusammenhang von der nun ungültigen Variable abhängen.

Das Ownership-Modell wird in der Praxis von Compiler durchgesetzt. So werden die Garantien, welche das Ownership-Modell verspricht, bereits zur Compile-Zeit sichergestellt. Zusätzlich fällt die Notwendigkeit für einen Garbage Collector weg, welches zu einer besseren Laufzeitseffizienz und Determinismus beitragen.

Typsystem

Rust verfügt über ein statisches, typsicheres Typsystem. Dies bedeutet dass jede Variable einen eindeutigen Typ besitzt und keinen anderen Typ annehmen kann.

Typsystem - Hindley-Milner

Typsicher

Typinferenz

Lineare Typen / -> Affine Typen <-

Generics

Paradigmen

Mehrere Pardigmen (OOP, Funktional)

Die Programmiersprache Rust lehnt sich syntaktisch an andere C-artige Sprachen an, unterscheidet sich aber hierbei in einigen Aspekten. So sind in ‘if‘/‘else‘ Blöcken beispielsweise die runden Klammern beispielsweise Optional, so wie es auch in Sprachen wie Python der Fall ist.

Fehlerbehandlung

Keine Exceptions -> Result<T,U>

Keine Race Conditions - Sync/Send

2.1.3 Architektur/Compiler

Der offizielle Rust Compiler rustc

Als Lösung zum Abhängigkeitsmanagement und der Distribution wurde das Tool ‘cargo‘ entwickelt. Es ermöglicht es dem Programmierer verwendete Bibliotheken in ein Projekt einzugliedern, indem diese in einer ‘.toml‘ Datei im Hauptverzeichniss des Projekts angegeben werden. Außerdem unterstützt ‘cargo‘ mehrere weiter hilfreiche Funktionen zum testen oder distributieren der entwickelten Software. Kompilierte Software kann als ein sogenanntes ‘Crate‘ bei der von den Rust entwickelten Platform[Citation Needed] ‘crates.io‘ mithilfe von ‘cargo‘ hochgeladen werden.

Etwas zum Cross Compiling

2.2 SPARC

Sparc

2.3 Invasives Computing

Invasives Computing ist ein paralleles Programmiermodell, welches es ermöglicht, temporär Ressourcen auf einem Parallelrechner zu beanspruchen und anschließend wieder freizugeben.Hierbei gibt es drei wesentliche Phasen, die ‘invade‘, ‘infect‘ und ‘retreat‘ Phasen.

In der ‘invade‘ Phase werden zunächst Ressourcen für das laufende Programm reserviert. Welche Ressourcen genau reserviert werden, wird durch ‘constraints‘ bestimmt.

Anschließend wird in der ‘infect‘ Phase die Funktion auf den reservierten Prozessorelementen ausgeführt.

Werden die reservierten Ressourcen nicht mehr benötigt, so kann man diese in der ‘retreat‘ Phase wieder freigeben.

3 Entwurf und Implementierung

Im folgenden wird die Implementierung des octorust Compilers und der octolib Bibliothek erläutert, welche den Gebrauch von Rust im invasiven Computing ermöglichen.

3.1 Rust auf der SPARCv8 Platform

Rust wird derzeit nicht offiziell auf der SPARCv8 Platform unterstützt, da Rust jedoch als Backend LLVM verwendet und dieses die Platform unterstützt, ist es auch möglich Rust auf dieser Architektur auszuführen.

Rust erlaubt es, Programme ohne die Standardbibliothek zu implementieren, welches in diesem Fall notwendig ist, da die Standardbibliothek nicht trivial auf jeder beliebigen Platform verwendbar ist.

nostd core

rustc json

cargo json

3.2 Erstellung des octorust Compilers

Um das komplizierte Vorgehen bei der “nostd” Kompilierung und das anschließende Verlinken mit der IRTSS Runtime zu vereinfachen, wurde ein Python Programm namens “octorust” implementiert. Dieses erlaubt es,

struktur

linken

3.3 Bindings zur C API

Im folgenden wurde eine Rust Bibliothek geschrieben, welche

3.4 Rust-spezifische Verbesserungen

AgentClaim / Constraints Structs

infect

Implizites retreat

Closures

4 Evaluation

4.1 Vergleich mit C

4.2 Vergleich mit X10

5 Fazit und Ausblick

Fazit!

Literaturverzeichnis

[1] stackoverflow.com, “Developer survey results 2016,” 2016.

[2] wikipedia.org, “Servo (Software),” 2017.

[3] debian.org, “Rust programs versus C gcc,” 2017.

[4] debian.org, “Rust programs versus Java,” 2017.

[5] debian.org, “Rust programs versus Python 3,” 2017.

Erklärung

Hiermit erkläre ich, Hermann Krumrey, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Danke

Ich danke meinen Eltern, die mich meine gesamtes Leben lang durch dick und dünn begleitet und unterstützt haben. Ich danke meinem Bruder, in dem ich einen ewigen Kumpanen im Leben gefunden habe. Ich danke meinem Betreuer Andreas Zwinkau, welcher mich freundlich und hilfreich durch die Erstellung dieser Arbeit begleitet hat. Außerdem danke ich meinen guten Freunden Simon Eherler und Frederick Horn, ohne die ich nicht der Mensch wäre der ich heute bin. Ich danke meinen Kommilitonen Marius Take, Johannes Bucher, Thomas Schmidt und Daniel Mockenhaupt, ohne die das Studium nicht halb so schön wäre. Ich danke meiner Ërsatzfamilie", der Familie Eherler, die immer einen Platz in ihrer Mitte für mich haben. Und zu guter Letzt danke ich dem Karlsruher Institut für Technologie, welches es mir erst ermöglichte, dieses Studium zu absolvieren.