Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

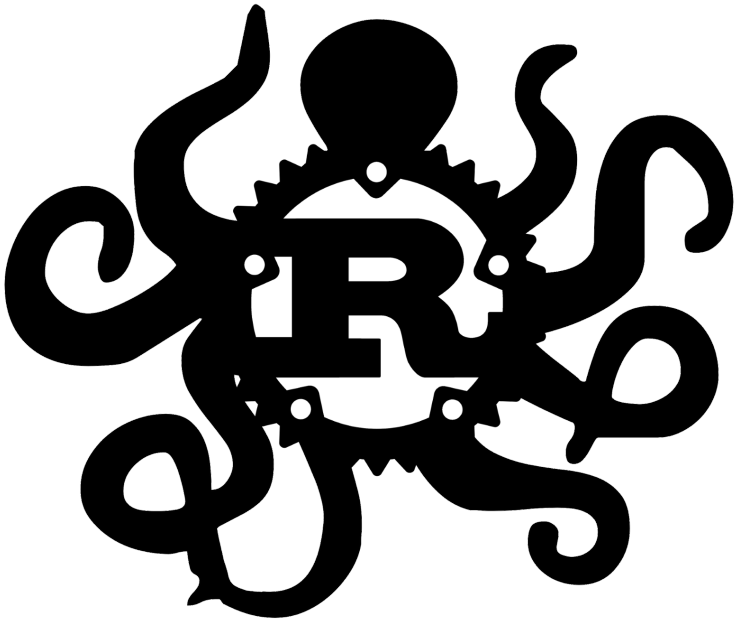
Lehrstuhl Prof. Dr.-Ing. Snelting

Invasives Rust

Bachelorarbeit von

Hermann Heinz Erich Krumrey

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuende Mitarbeiter:	Dipl.-Inform. Andreas Zwinkau

Bearbeitungszeit: 3. Juli 2017 – 2. November 2017

Zusammenfassung

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

Das *OctoPOS* Betriebssystem und die invasive Middleware *iRTSS*, welche eine beispielhafte Implementierung für ein solches invasives System bietet, unterstützen derzeit die Verwendung der Programmiersprachen C, C++ und X10. Hiermit wird die Einführung der Programmiersprache Rust als weitere unterstützte Sprache vorgeschlagen.

Im Rahmen dieser Arbeit wurde das *octorust* Programm und die zugehörige *octolib* Bibliothek entwickelt, welche es ermöglichen, Rust auf invasiven Systemen zu verwenden. Der Einsatz von Rust ermöglicht das speichersichere Programmieren ohne einen *Garbage Collector*, wobei in Folge dessen keine Kompromisse bezüglich der Laufzeit eingegangen werden müssen. Daher bietet Rust einen Vorteil gegenüber den bereits von *OctoPOS* und *iRTSS* unterstützten Sprachen.

Vorläufige Messungen haben ergeben, dass Rust in Situationen, in denen häufige Speicherallokationen und -deallokationen vorkommen, eine über zehn mal kürzere Laufzeit als X10 aufweisen kann. Bei mathematischen Berechnungen hingegen weist Rust zumeist ein schlechteres Laufzeitverhalten als C oder X10 auf.

Inhaltsverzeichnis

1 Einführung **7**

2 Grundlagen **9**
2.1 Rust . . . 9
2.1.1 Motivation . . . 9
2.1.2 Das Typsystem . . . 10
2.1.3 Fehlerbehandlung . . . 13
2.1.4 Interaktion mit anderen Programmiersprachen . . . 13
2.1.5 `unsafe`-Blöcke in Rust . . . 14
2.1.6 Architektur/Compiler . . . 14
2.2 SPARC und SPARC-V8 . . . 15
2.2.1 LEON . . . 15
2.3 Invasives Rechnen . . . 16
2.4 Verwandte Arbeiten . . . 16

3 Entwurf und Implementierung **17**
3.1 Rust auf der SPARC LEON Architektur . . . 17
3.2 Erstellung des *octorust* Hilfsprogramms . . . 19
3.2.1 Struktur eines invasiven *Cargo*-Projekts . . . 21
3.2.2 Kompilieren . . . 21
3.3 *octolib* . . . 22
3.3.1 Struktur . . . 23
3.3.2 Direkte C-Rust-*Bindings* . . . 23
3.3.3 Rust-spezifische Verbesserungen . . . 24

4 Evaluation **29**
4.1 Kompilierungsdauer und Dateigröße . . . 31
4.2 Laufzeitverhalten . . . 32
4.2.1 Vergleich der Anlaufzeit . . . 32
4.2.2 Berechnen von Primzahlen . . . 33
4.2.3 Paralleles Berechnen von Primzahlen . . . 34
4.2.4 Allokation auf dem Heap . . . 37
4.3 Aufwand des Programmierens . . . 38
4.3.1 Projektstruktur . . . 38
4.3.2 Minimales Invade, Infect, Retreat . . . 39

5 Fazit und Ausblick **45**
5.1 Fazit 45
5.2 Ausblick 46

1 Einführung

Gordon Moore hat im Jahre 1965 das Mooresche Gesetz formuliert, welches besagt, dass sich die Integrationsdichte von integrierten Schaltkreisen in regelmäßigen Intervallen verdoppelt [1]. Diese Beobachtung erwies sich bisher als korrekt, allerdings wird es nach und nach wahrscheinlicher, dass die heutigen Fertigungsmethoden diesen Trend nicht unendlich fortführen können, da diese an physische Grenzen stoßen werden [2].

Um zukünftig trotzdem eine verbesserte Rechenleistung zu erzielen, wird unter anderem auf Parallelrechner gesetzt. Der Grundgedanke dahinter ist es, mehrere Recheneinheiten zu verwenden, welche gemeinsam eine Aufgabe lösen. Somit wird keine Verbesserung der einzelnen Recheneinheiten benötigt, um eine verbesserte Gesamtrechenleistung zu erzielen.

Der Trend zum parallelen Rechnen kann bereits heute beobachtet werden. So sind moderne PCs oder Smartphones generell alle mit Mehrkernsystemen ausgestattet. Bei Grafikprozessoren kommen mittlerweile bereits Tausende einzelner Rechenkerne zum Einsatz, beispielsweise verfügt die Nvidia GTX 1080 GPU laut Spezifikation [3] über 2560 Kerne.

Durch den Einsatz von parallelen Rechnern entstehen jedoch auch Kosten für den Programmierer, denn das Rechnersystem wird hierdurch komplexer. Während der Entwicklung muss sichergestellt werden, dass durch die gleichzeitige Verarbeitung durch die einzelnen Recheneinheiten keine Fehler entstehen. Die individuellen Prozessorkerne müssen also miteinander kommunizieren und die Hardwareressourcennutzung untereinander koordinieren.

Um die zusätzliche Komplexität des parallelen Rechnens für den Programmierer zu verringern, müssen neue Techniken oder Programmierparadigmen entwickelt werden. Eine solche wäre das invasive Rechnen, welches es einem Programmierer erlaubt, die Ressourcen in parallelen Systemen besser zu nutzen. Vor allem bei Systemen mit vielen Rechenkernen ist dieses Paradigma ein interessanter Lösungsansatz. So kann man beispielsweise auf einer Nvidia GTX 1080 GPU mit insgesamt 2560 Kernen ein Programm ausführen, welches ein Problem bearbeitet, das auf genau 1000 dieser Kerne ausgeführt wird. Gleichzeitig kann ein anderes Problem auf einer weiteren Untermenge der Kerne ausgeführt werden.

Die derzeit existierenden invasiven Systeme unterstützen nur die Programmiersprachen C, C++ und X10. Eine interessante Addition hierzu wäre die Sprache Rust, welche einen Fokus auf die Sicherheit vor schwerwiegenden Programmierfehlern, die vor allem durch Zugriffe auf ungültige Speicherregionen ausgelöst werden, legt.

Damit Rust im Kontext des invasiven Rechnens verwendet werden kann, werden zunächst die Grundlagen der Programmiersprache Rust, des invasiven Rechnens und der Prozessorarchitektur SPARC-V8 erarbeitet. Anschließend werden Werkzeuge zum Erstellen von invasiven Rust-Programmen entwickelt, welche das Ziel verfolgen, Rust effektiv in Verbindung mit diesem Programmierparadigma zu verwenden. Abschließend muss der tatsächliche Wert dieses Unterfangens evaluiert und bewertet werden.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen der Programmiersprache Rust, der Rechnerarchitektur SPARC-V8 und des invasiven Rechnens behandelt.

2.1 Rust

Rust ist eine Programmiersprache, welche wie beispielsweise C++ direkte Kontrolle über die unterliegende Hardware bieten soll. Dies bedeutet in der Praxis, dass diese Sprache nicht von einem *Garbage Collector* Gebrauch macht, wie es beispielsweise bei der Sprache X10 der Fall ist. Im Gegensatz zu C++ bietet Rust jedoch standardmäßig automatische Speichersicherheit und garantiert somit, dass Programme keine Speicherfehler enthalten [4].

Rust erfreut sich seit ihrer Veröffentlichung wachsender Beliebtheit bei Programmierern aller Art. Sie wurde bei einer Umfrage der Webseite *stackoverflow.com* in den Jahren 2016 und 2017 als beliebteste Programmiersprache bei Entwicklern ermittelt [5][6].

Das wohl derzeit prominenteste Projekt, welches Rust verwendet, ist der *Servo Web Browser Engine*. Dieses Projekt wurde von Mozilla Research initiiert, erhält jedoch ebenfalls Beiträge von Freiwilligen als auch von Unternehmen wie Samsung. Diese Software soll nach und nach im *Mozilla Firefox Webbrowser* integriert werden und hierbei eine bessere Leistung und Sicherheit als vorhergehende Technologien vorweisen [7].

2.1.1 Motivation

Eines der Kernziele der Rust Programmiersprache ist es, sichere Speicherzugriffe zu gewährleisten, ohne für diesen Zweck einen *Garbage Collector* zu verwenden. Fehlerhafte Speicherzugriffe können undefiniertes Verhalten auslösen, welches ein Sicherheitsrisiko darstellen kann, denn diese Fehler können gezielt ausgenutzt werden,

um Schadcode auszuführen. Hierbei beziehen sich fehlerhafte Speicherzugriffe auf das Verwenden von Speicherregionen, die entweder ungültig sind oder bereits befreit wurden [7][8].

Außerdem soll Rust paralleles Rechnen unterstützen und so möglichst effizient moderne Hardwarearchitekturen ausnutzen. Hierfür sollen gewisse häufig auftretende Fehler in der Parallelprogrammierung gänzlich entfallen [4].

Rust lehnt sich konzeptionell und syntaktisch an C-ähnliche Sprachen an, enthält jedoch auch Konzepte aus der funktionalen Programmierung. Zusätzlich werden Sprachkonzepte und Abstraktionen geboten, die ansonsten zumeist in höheren Programmiersprachen vorhanden sind. Hierdurch soll der Einstieg für Programmierer, die zuvor keine oder nur wenige Erfahrungen mit Systemsprachen gesammelt haben, erleichtert werden [7].

Diese zusätzlichen Sicherheiten und Nutzerfreundlichkeiten sollen gleichzeitig jedoch keine Laufzeitkonsequenzen mit sich ziehen, denn es ist ein Ziel von Rust, eine ähnliche Recheneffizienz im Vergleich mit bestehenden Systemsprachen wie C oder C++ vorzuweisen. Außerdem soll Rust es vereinfachen, mehrere Programmiersprachen miteinander zu verwenden. Das Ziel hierbei ist es, Code anderer Sprachen aus einem Rust Kontext heraus zu verwenden, als auch umgekehrt [9].

2.1.2 Das Typsystem

Das signifikanteste Alleinstellungsmerkmal der Programmiersprache Rust ist ihr Typsystem. Dieses ermöglicht es erst, die versprochenen Sicherheitsgarantien ohne zusätzliche Konstrukte wie einem *Garbage Collector* zu realisieren [9].

Ownership, Move-Semantik und Borrowing

Das Typsystem integriert das Konzept der *Ownership (engl. Besitz)*. Dies ist die zentrale Besonderheit von Rusts Typsystem. Der Grundgedanke hierhinter ist es, dass der Zugriff auf eine Speicherregion exklusiv einer Variable zur Verfügung gestellt wird. Diese Variable wird auch als *Owner (engl. Besitzer)* bezeichnet. Zu jedem Zeitpunkt darf die Speicherregion nur einen *Owner* haben. Sobald dieser *Owner* den Geltungsbereich verlässt, wird die zugehörige Speicherregion freigegeben, die dort vorhandenen Daten können anschließend nicht mehr verwendet werden. Dies ist der Mechanismus, der es in Rust erlaubt, Speicher ohne manuelle Befreiung oder Nutzung eines *Garbage Collectors* zu verwalten. Er stellt sicher, dass keine fehlerhaften Speicherzugriffe durch die Nutzung von ungültigen Daten im Speicher

Listing 2.1: Beispieldarstellung der *Move*-Semantik

```
let x = String::from("hello");
// Owner: x

let y = x;
// Owner: y
// x ist nun ungültig

let z = f(y)
// Owner: -
// Ownership wurde an f übergeben,
// da f aber bereits den Geltungsbereich verlassen hat
// ist der Speicherbereich nun befreit.
// y ist nun ungültig
```

entstehen [9]. Außerdem können so Speicherlecks vermieden werden. Das Konzept des *Ownership* wurde von affinen Typen inspiriert.[10] Solche Typen können maximal einmal verwendet werden [11].

Die *Ownership* einer Speicherregion kann mithilfe der *Move*-Semantik den Besitzer wechseln. Dies kann bei Funktionsaufrufen oder auch bei Zuweisungen geschehen. Beispiele hierfür sind in Listing 2.1 zu sehen [9].

Zusätzlich zur *Move*-Semantik können in Rust Referenzen zu Variablen verwendet werden. Dies wird im Kontext von Rust häufig als *borrowing (engl. ausleihen)* bezeichnet. Verwendet man eine Referenz auf eine Variable, ist der *Owner* des Speicherbereichs weiterhin die originale Variable. Zur Compile-Zeit wird überprüft, dass alle Referenzen gültig sind, um so die Existenz von hängenden Zeigern zu verhindern. Eine Referenz kann also nie auf eine Variable zeigen, die bereits den Geltungsbereich verlassen hat [9].

Es gibt zwei unterschiedliche Typen von Referenzen, veränderliche und unveränderliche. Standardmäßig werden in Rust unveränderliche Referenzen verwendet. Diese erlauben es nicht, den Wert des Speicherbereichs zu verändern, sondern nur auszulesen. Es gibt keine Limitierung bei der Anzahl unveränderlicher Referenzen die auf eine Variable zeigen können. Im Gegensatz hierzu kann nur eine einzelne veränderliche Referenz auf eine Variable existieren. Dies ermöglicht es, bereits zur Compile-Zeit gewisse Wettlaufsituationen, sogenannte *Data Races*, in der Parallelprogrammierung auszuschließen [9].

Strukturen, Implementierungen, Aufzählungen und Traits

Zum Erstellen von benutzerdefinierten Typen bietet Rust Strukturen (`struct`), Implementierungen (`impl`), Aufzählungen (`enum`) und Traits (`trait`).

Strukturen in Rust spezifizieren benutzerdefinierte Datentypen, welche verwandte Daten vereinen. Das Konzept ähnelt den Strukturen in C. Anders als in C kann man jedoch mithilfe von Implementierungen Methoden und assoziierte Funktionen für diese Strukturen definieren. Diese Möglichkeit erlaubt es, eine objektorientierte Herangehensweise in Rust zu verwenden. Implementierungen erlauben das Definieren von Methoden, welche auf einer initialisierten Struktur aufgerufen werden können und eine Referenz auf `self`, also der Struktur selbst, als Parameter entgegennehmen. Zusätzlich können Funktionen definiert werden, welche keinen solchen Parameter entgegennehmen und somit auch mit Strukturen verwendet werden können, ohne diese im Voraus zu initialisieren. Dies ähnelt statischen Methoden in objektorientierten Sprachen [9].

Aufzählungen agieren in Rust ähnlich zu anderen Programmiersprachen. Der Vorteil der Aufzählungen in Rust ist es, dass diese für *Pattern Matching* verwendet werden können. Ein Beispiel dazu, wie dies in der Praxis verwendet werden kann, wird in Kapitel 2.1.3 beschrieben [9].

Traits sind Sprachkonstrukte in Rust, welche das Verhalten von Typen abstrakt definieren können. Sie ähneln hierbei Schnittstellen (*Interfaces*) aus anderen Programmiersprachen. Zusätzlich zur benutzerdefinierten Spezifikation solcher *Interface*-ähnlichen Konstrukte bietet Rust vordefinierte Traits, welche das Verhalten der Typen bezüglich *Ownership* und *Move*-Semantik beeinflussen können [9].

Ein solches Trait ist das `Drop`-Trait. Dieses erlaubt es, das Verhalten eines Typen beim Verlassen des Geltungsbereichs zu beeinflussen. In der `drop`-Methode des Traits wird es ermöglicht, einen Destruktor für den Typ zu definieren. Dieser kann aus unterschiedlichen Gründen benötigt oder hilfreich sein. Beispielsweise kann ein solcher Destruktor verwendet werden, um einen Referenzzähler zu dekrementieren, unbenutzte Netzwerkverbindungen zu trennen oder anderweitig Ressourcen wieder freizugeben [9].

Außerdem gibt es die `Copy`-Traits, welche es ermöglichen, anstelle der *Move*-Semantik die *Copy*-Semantik zu verwenden. Für Typen, die dieses Trait implementieren, wird jedes mal, wenn ein Speicherbereich im Kontext der *Move*-Semantik den *Owner* wechseln würde, stattdessen eine bitweise Kopie der Daten erstellt. Primitive Datentypen wie `i32` verhalten sich standardmäßig bereits so. Das `Copy`-Trait kann jedoch nicht auf jeden Typ angewandt werden. Insbesondere kann ein Typ nicht das `Copy`- als auch das `Drop`-Trait implementieren, da Typen die das `Drop`-Trait implemen-

tieren beim Verlassen des Geltungsbereichs einen Einfluss außerhalb ihres eigenen Speicherbereichs haben können. Würde der Destruktor für jede Kopie aufgerufen werden, könnte dies zu unvorhergesehenen und unerwünschten Konsequenzen führen [9][12].

2.1.3 Fehlerbehandlung

Obwohl der Compiler dank Rusts Typsystem bereits bei der Kompilierung eine Vielzahl von Fehlern erkennen kann, ist dies nicht für alle Fehlertypen möglich. Beispielsweise kann es vorkommen, dass durch inkorrekte Indizierung eines Arrays ein Fehler entsteht. Solche Fehler müssen zur Laufzeit behandelt werden [9].

Fehler werden in Rust idiomatisch mithilfe von Rückgabewerten vom Aufzählungstyp `Result` behandelt. Anhand dieser Rückgabewerte lässt sich mithilfe von *Pattern Matching* prüfen, ob ein Fehler beim Funktionsaufruf aufgetreten ist. Ist keiner aufgetreten, kann der eigentliche Rückgabewert der Funktion aus dieser Aufzählung entnommen werden [9]

Wird ein Fehler nicht auf diese Art und Weise behandelt, wird das `panic!`-Makro aufgerufen. Dieses befreit standardmäßig die Daten des Programms auf dem Stack, gibt anschließend einen *Stacktrace* aus und beendet darauf die Ausführung des Programms. Es ist jedoch auch möglich, ein Programm so zu konfigurieren, dass stattdessen die Ausführung des Programms beendet wird sobald ein unbehandelter Fehler auftritt [9].

2.1.4 Interaktion mit anderen Programmiersprachen

Eine praktische Eigenschaft Rusts ist es, dass sie in Verbindung mit einer Vielzahl von anderen Programmiersprachen verwendet werden kann. Mithilfe des *Foreign Function Interface* kann man Code aus anderen Sprachen in Rust verwenden. Dadurch kann Rust beispielsweise Bibliotheken, welche in C geschrieben sind, verwenden. Das *Foreign Function Interface* erlaubt es zusätzlich, Rust-Code aus dem Kontext anderer Sprachen zu verwenden. Dies wird für eine Vielzahl an Programmiersprachen unterstützt, unter anderem C, aber auch höheren Programmiersprachen wie Python oder Ruby, welche durch die Einbindung von Rust-Code Leistungssteigerungen erzielen können [13][14].

2.1.5 `unsafe`-Blöcke in Rust

In gewissen Situationen kann es sein, dass Rusts Sicherheitsgarantien die Implementierung erschweren oder sogar unmöglich machen. Vor allem bei sehr Hardware-nahen Programmen, beispielsweise einem Betriebssystem, kann eine solche Situation vorkommen. Als Lösung für dieses Problem bietet Rust **`unsafe`**-Blöcke an. Innerhalb dieser ist es möglich, Zeiger zu dereferenzieren, welche direkt ein Speichersegment adressieren. Somit hat man direkte Kontrolle über diesen Speicher, was jedoch bei Unachtsamkeit seitens des Entwicklers zu Fehlern oder undefiniertem Verhalten führen kann. Außerdem können unsichere Funktionen, beispielsweise solche die mithilfe des *Foreign Function Interface* aus anderen Programmiersprachen importiert wurden, nur in **`unsafe`**-Blöcken verwendet werden [9].

Auch in **`unsafe`**-Blöcken bleibt das Konzept von *Ownership* und die zugehörige *Move*-Semantik und *Borrowing* bestehen. Diese Sicherheitsmechanik wird also nicht aufgegeben wenn man die Funktionalität der **`unsafe`**-Blöcke verwendet [9].

2.1.6 Architektur/Compiler

Der offizielle Rust Compiler heißt **`rustc`** und kann eigenständige Rust-Quelldateien kompilieren. Rust-Quelldateien haben per Konvention die Dateiendung **`.rs`** [9].

Als Lösung zum Abhängigkeitsmanagement und der Distribution wurde das Werkzeug **`cargo`** entwickelt. Dieses ermöglicht es dem Programmierer, verwendete Bibliotheken in ein Projekt einzugliedern, indem diese in einer **`Cargo.toml`**-Konfigurationsdatei im Hauptverzeichnis des Projekts angegeben werden. Hierbei ist es möglich, Bibliotheken von der zentralen *crates.io* Plattform automatisch herunterladen zu lassen oder einen lokalen Dateipfad zu einer solchen Bibliothek anzugeben. Außerdem unterstützt **`cargo`** hilfreiche Funktionen zum Testen oder Verteilen der entwickelten Software. Entwickelte Software kann als ein sogenanntes *Crate* bei *crates.io* hochgeladen werden [9].

Um den Gebrauch von unterschiedlichen **`rustc`** und **`cargo`** Versionen zu vereinfachen, wurde das Werkzeug **`rustup`** entwickelt. Dieses Hilfsprogramm ermöglicht es, unterschiedliche Varianten des Rust-Compilers zu installieren und bei Belieben zu wechseln. Es bietet auch die Funktionalität, die benötigten Kern- und Standardbibliotheken für unterstützte Architekturen herunterzuladen, um Programme für andere Zielarchitekturen zu kompilieren [15].

Sowohl **`rustc`** als auch **`cargo`** ermöglichen das Kompilieren für unterschiedliche Zielarchitekturen mithilfe der **`--target`**-Option. Als Compiler-Backend wird *LLVM*

verwendet. Rust-Code wird also vor der Übersetzung zu Maschinenbefehlen in die Zwischensprache *LLVM-IR* übersetzt und erst anschließend weiterverarbeitet. Somit ist es möglich, Rust-Programme für alle von *LLVM* unterstützten Rechnerarchitekturen zu kompilieren [16].

2.2 SPARC und SPARC-V8

SPARC (**S**calable **P**rocessor **ARC**hitecture) ist eine Rechnerarchitektur, welche auf das RISC (Reduced Instruction Set Computing) Konzept aufbaut. Eines der Ziele der SPARC-Architektur ist es, skalierbar zu sein und so in unterschiedlichsten Umgebungen zum Einsatz zu kommen. So können SPARC-Prozessoren in Mikrocontrollern bis hin zu Supercomputern verwendet werden. Die Architektur wird an unterschiedliche Halbleiter-Unternehmen lizensiert, mit dem Ziel durch konkurrierende Kräfte Innovation bezüglich Kosten und Rechenleistung zu fördern [17].

SPARC-V8 ist eine auf SPARC basierende Architektur. Es handelt sich hierbei um eine Allzweckarchitektur mit einem 32-bit breiten Datenpfad [18].

2.2.1 LEON

Die ursprünglich von der European Space Agency (ESA) und anschließend von Gaisler Research entwickelten LEON-Prozessoren basieren auf der SPARC-V8-Architektur. Das Hauptziel der LEON-Prozessoren war es, fehlertolerante Nachfolger zu den älteren SPARC-V7-basierten Prozessoren, die zu dem Zeitpunkt bei der ESA Verwendung fanden, zu bieten [19]. Es handelt sich bei der LEON-Prozessorfamilie um stark konfigurierbare Implementierungen der SPARC-V8-Architektur in der Hardwarebeschreibungssprache VHDL (*Very High Speed Integrated Circuit Hardware Description Language*). Unterschiedliche Iterationen der Prozessorfamilie sind unter Open-Source Lizenzen als VHDL-Designs verfügbar. So wurden die VHDL-Designs des LEON2 Prozessors unter der GNU LGPL Lizenz veröffentlicht [18] und auch das Design des LEON3 Prozessors ist unter einer Open-Source Lizenz verfügbar [20].

Die Kombination von stark konfigurierbaren Designs und der Open-Source Lizenz ermöglicht den Gebrauch der LEON VHDL-Designs in domänenspezifischen, angepassten *ASICs (Application-Specific Integrated Circuit)*, *FPGAs (Field Programmable Gate Array)* oder *SOCs (System On Chip)*. Daher eignet sich die LEON Prozessorfamilie auch zum Einsatz im invasiven Rechnen, um deren neuartigen Konzepte möglichst effizient durch angepasste Hardware auszunutzen [18].

2.3 Invasives Rechnen

Invasives Rechnen beschreibt die Möglichkeit eines Programms auf einem Parallelrechner, dynamisch Hardwareressourcen zu reservieren, nutzen und anschließend wieder freizugeben [21].

Die drei fundamentalen Phasen des invasiven Rechnens sind die ***Invade***-, ***Infect***- und ***Retreat***-Phasen. In der *Invade*-Phase werden zunächst die gewünschten Hardwareressourcen angefordert und anschließend reserviert. Diese Menge an Ressourcen kann dann mithilfe eines *Claims* verwaltet werden. Die zu reservierenden Ressourcen werden mithilfe von *Constraints* spezifiziert. Wurden die Ressourcen erfolgreich reserviert, kann die *Infect* Phase beginnen. In dieser werden Ausführungsfäden erstellt, welche auf den reservierten Hardwareressourcen ausgeführt werden. Im Kontext des invasiven Rechnens wird ein solcher Faden als *invasive-let* oder abgekürzt als *i-let* bezeichnet [22]. Benötigt das Programm die reservierten Ressourcen nicht mehr, können diese in der *Retreat*-Phase wieder freigegeben werden [21].

Für den praktischen Einsatz des invasiven Rechnens wurde ein Betriebssystem namens *OctoPOS* entwickelt. Dieses unterstützt die *Invade*-, *Infect*- und *Retreat*-Phasen auf der Systemebene. Zusätzlich hierzu existiert das *invasive Run-Time Support System (iRTSS)*. Diese Middleware fungiert als eine Hardware-Abstraktionsschicht und als Ressourcenverwalter. *OctoPOS* und *iRTSS* bieten so eine Plattform für das invasive Programmieren und unterstützen derzeit den Gebrauch der Programmiersprachen C, C++ und X10 [21][23][24].

Um das Konzept des invasiven Rechnens möglichst effizient auszunutzen, wird speziell an das invasive Rechnen angepasste Hardware entwickelt, deren Prozessorkerne auf dem Design des LEON3-Prozessors basieren. Diese Hardware unterstützt gewisse invasive Grundfunktionen und entlastet somit das *OctoPOS* Betriebssystem und das *iRTSS* [20].

2.4 Verwandte Arbeiten

Eine verwandte Arbeit ist „Invasive Computing—An Overview“ [25] von Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat und Gregor Snelting. Diese illustriert die Grundkonzepte des invasiven Rechnens. Eine weitere verwandte Arbeit ist „Technical Report: An X10 Compiler for Invasive Architectures“ [26] von Sebastian Buchwald, Manuel Mohr und Andreas Zwinkau, welche die Erstellung eines X10 Compilers für den Gebrauch im invasiven Rechnen beschreibt.

3 Entwurf und Implementierung

Im folgenden werden Programme und Bibliotheken entwickelt, welche es ermöglichen, Rust auf der von *OctoPOS* und *iRTSS* zur Verfügung gestellten invasiven Plattform verwenden zu können. Dies ermöglicht es Programme, welche vom invasiven Rechnen Gebrauch machen, in der Programmiersprache Rust zu schreiben.

3.1 Rust auf der SPARC LEON Architektur

Rust wird derzeit nicht offiziell auf der SPARC-V8-Architektur unterstützt. Rust verwendet jedoch als Compiler-Backend LLVM, das für diese Architektur Maschinenbefehle aus der Zwischensprache LLVM-IR generieren kann. Daher ist es prinzipiell möglich, Rust-Programme für diese Architektur zu kompilieren. Die Rust-Standardbibliothek ist allerdings schwierig auf andere Architekturen zu portieren. Um diese Hürde zu umgehen bietet Rust die Funktion, Programme mit einer minimalen, plattformunabhängigen Untermenge der Standardbibliothek zu kompilieren. Diese Funktionalität wird hier ausgenutzt, um eine minimale Implementierung der Programmiersprache auf die SPARC-V8-Architektur zu portieren.

Um ein Rust-Programm für eine nicht offiziell unterstütze Architektur zu kompilieren, muss man zuerst die *libcore*-Bibliothek für die Zielarchitektur kompilieren. Um dies zu bewerkstelligen, benötigt man eine JSON-Datei, welche dem Compiler die nötigen Informationen bezüglich der Zielarchitektur zur Verfügung stellt. Eine solche JSON-Datei für die SPARC-V8 Architektur wird beispielsweise in Listing 3.1 veranschaulicht [27].

Außerdem wird ein Linker, beispielsweise `gcc`, für die SPARC-V8 Architektur benötigt. Der Pfad zu diesem Linker muss in der `linker`-Option der JSON-Datei angegeben werden.

Sobald man diese JSON-Datei erstellt hat, kann man die *libcore*-Bibliothek mit dem `cargo build`-Befehl kompilieren. Um dies für SPARC-V8 zu bewerkstelligen, verwendet man als Parameter für die `--target`-Option den Pfad zur JSON-Spezifikationsdatei. Derzeit ist es nicht möglich, *libcore* mit dem stabilen Rust-

Listing 3.1: Eine beispielhafte JSON-Spezifikationsdatei für die SPARC-V8 Architektur [?]

```
{
    "arch": "sparc",
    "data-layout": "E-m:e-p:32:32-i64:64-f128:64-n32-S64",
    "executables": true,
    "llvm-target": "sparc",
    "os": "none",
    "panic-strategy": "abort",
    "target-endian": "big",
    "target-pointer-width": "32",
    "linker-flavor": "ld",
    "linker": "Pfad zum SPARC-V8 Linker",
    "link-args": [
        "-nostartfiles"
    ]
}
```

Compiler zu kompilieren, da diese Bibliothek Funktionalitäten verwendet, welche auf dem stabilen Compiler deaktiviert sind. Daher muss ein *Nightly*-Compiler installiert werden, was dank **`rustup`** benutzerfreundlich gestaltet ist. Außerdem sollte die Version der *libcore*-Bibliothek mit der Version des *Nightly*-Compilers übereinstimmen, um versionsabhängige Konflikte bei der Kompilierung zu vermeiden.

Nachdem die *libcore*-Bibliothek erfolgreich kompiliert wurde, kann man die resultierende **`.rlib`**-Datei anderen Rust-Programmen beim Kompilieren zur Verfügung stellen, indem man diese in das korrekte Unterverzeichnis im lokalen **`~/.rustup`** Verzeichnis kopiert. Alternativ kann man das *libcore Cargo*-Projekt direkt durch die Abhängigkeiten in der **`Cargo.toml`** Datei eines *Cargo*-Projekts einbinden.

Soll ein Rust-Programm *libcore* anstelle der Standardbibliothek verwenden, muss man dies mit der Anweisung **`#![no_std]`** zu Beginn des Programms kenntlich machen. Außerdem müssen die Funktionen **`eh_personality`**, **`eh_unwind_resume`** und **`panic_fmt`** in diesem Programm manuell implementiert werden. Nennenswert ist vor allem Letztere, denn diese Funktion wird aufgerufen, sobald ein Programm nach einem unbehandelten Fehler kontrolliert die Ausführung beendet. Der Anfang eines zu kompilierenden Programms muss also aussehen wie das Beispiel in Listing 3.2.

Um anschließend das Programm zu kompilieren, verwendet man entweder **`rustc`** bei eigenständigen **`.rs`** Dateien oder den **`cargo build`**-Befehl für *Cargo*-Projekte. Beiden Befehlen muss wie auch beim Kompilieren von *libcore* die JSON-Spezifikationsdatei

Listing 3.2: Der Beginn eines Rust-Programms, welches nicht die Standardbibliothek verwendet.

```
#![feature(lang_items, libc)]
#![no_std]
#![no_main]

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "eh_unwind_resume"] extern fn eh_unwind_resume() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

als Argument für die **`--target`**-Option übergeben werden.

3.2 Erstellung des *octorust* Hilfsprogramms

Um das Kompilieren für Rust-Programmen auf der SPARC-V8 Architektur zu vereinfachen, sowie besagte Rust-Programme zusammen mit dem *OctoPOS* Betriebssystem und dem *iRTSS* zu verwenden, wurde ein Python-Programm namens *octorust* geschrieben, welche diese Schritte vereinfacht. Das Programm wurde in Python geschrieben, da Pythons Standardbibliothek viele nützliche Funktionen zur Manipulation von Dateien bietet, welches sich für diesen Zweck als hilfreich erwiesen hat.

Das Programm verwendet **`python-setuptools`** und eine dafür konfigurierte **`setup.py`**-Datei, um das Programm lokal zu installieren. Während des Installationsprozesses wird zum einen das *octorust*-Programm lokal als Python-Modul installiert und das zugehörige **`octorust`**-Skript als ausführbare Datei dem Nutzer zur Verfügung gestellt. Gleichzeitig wird ein Verzeichnis namens **`.octorust`** im Heimverzeichnis des derzeitigen Nutzers erstellt. Dieses fungiert als Speicherort für *OctoPOS/iRTSS*-Builds, die in Kapitel 3.3 genauer beschriebene *octolib* Bibliothek und im Falle dass kein solcher zuvor installiert war, einen SPARC-V8 **`gcc`**. Zudem werden für alle unterstützten Architekturen die *libcore* und *libc* Bibliotheken kompiliert und in den Installationspfad der derzeit verwendeten **`rustup`**-Werkzeugkette kopiert.

Octorust bietet die folgenden Kommandozeilenoptionen:

-h, --help Diese Option gibt einen Nutzungshinweis inklusive aller möglichen Kommandozeilenoptionen aus.

-a. --architecture Diese Option ermöglicht es, eine *OctoPOS/iRTSS* Zielarchitektur

zu wählen. Zur Wahl stehen *x86guest*, *x64native* als auch *leon*. Sollte diese Option nicht angegeben werden, wird standardmäßig für die *x86guest* Architektur kompiliert.

-v, --variant Diese Option ermöglicht es, eine *OctoPOS/iRTSS*-Variante zu wählen. Sollte diese Option nicht angegeben werden, wird je nach gewählter Architektur eine passende Standardvariante verwendet. Im Falle von *x86guest* und *x64native* wird die Variante *generic* gewählt und für *leon* die Variante *4t5c-nores-chipit-w-iotile*.

-o, --output Mit dieser Option kann der Pfad zur resultierenden Ausgabedatei explizit angegeben werden. Ansonsten ermittelt *octorust* automatisch einen sinnvollen Ausgabenamen, beispielsweise `foo.out` für ein *Cargo*-Projekt mit dem Namen *foo*.

-k, --keep Wird diese Option verwendet, werden jegliche temporäre Dateien, die während dem Kompilieren und dem Linken erstellt werden, im Anschluss nicht gelöscht. Dies beinhaltet unter anderem erstellte statische Bibliotheken oder Objekt-Dateien.

-r, --run Wird diese Option verwendet, wird das Programm nach dem Kompilieren sofort ausgeführt. Für andere Architekturen als *x86guest* wird dabei vorausgesetzt, dass die Virtualisierungssoftware `qemu` installiert ist.

-i, --irtss-build-version Mithilfe dieser Option kann man eine spezifische *OctoPOS/iRTSS* Version verwenden.

--release Wird diese Option verwendet, werden Compiler-Optimierungen für Rust-Programme aktiviert.

--fetch-irtss Mithilfe dieser Option können *OctoPOS/iRTSS*-Builds von https://www4.cs.fau.de/invasic/octopos/ heruntergeladen werden. Hierfür wird jedoch eine gültige Nutzername/Passwort-Kombination in einer `~/.netrc` Datei im folgenden Format benötigt:

```
machine www4.cs.fau.de
login NUTZERNAME
password PASSWORT
```

Unterstützt wird das Kompilieren von eigenständigen C- und Rust-Quelldateien, als auch das Kompilieren von *Cargo*-Projekten. *Cargo*-Projekte bieten durch das Abhängigkeitsmanagement zusätzlich den Gebrauch von Rust-Bibliotheken, vor allem der *octolib*-Bibliothek, welche eigens für die Interaktion zwischen Rust und *OctoPOS/iRTSS* entwickelt wurde. Daher werden sich die folgenden Prozesse primär

Listing 3.3: Eine minimale Haupt-Quelldatei eines invasiven *Cargo*-Projekts

```
#![no_std]

extern crate octolib;

#[no_mangle]
pub extern "C" fn rust_main_ilet(claim: u8) {
}
```

mit dem Kompilierungsvorgang der *Cargo*-Projekte beschäftigen. Ein genauerer Überblick über die *octolib* Bibliothek folgt in Kapitel 3.3.

3.2.1 Struktur eines invasiven *Cargo*-Projekts

Wie normale *Cargo*-Projekte besteht ein invasives Cargo-Projekt aus mindestens einem `src`-Verzeichnis, einer Haupt-Bibliotheksdatei und einer `Cargo.toml`-Konfigurationsdatei. Bei invasiven *Cargo*-Projekten muss das `crate-type`-Attribut zudem immer als `staticlib` angegeben werden, damit das Projekt als statische Bibliothek kompiliert wird, welche dann mit *OctoPOS* und dem *iRTSS* verlinkt werden kann.

Zusätzlich muss die Haupt-Quelldatei des Projekts mit `#![no_std]` beginnen und anstelle der `main()` Funktion muss eine `pub extern "C"` Funktion namens `rust_main_ilet` als Startpunkt des Programms verwendet werden. Diese Funktion nimmt genau einen Parameter entgegen, welcher vom Typ `u8` ist. Vor dieser Funktion muss außerdem die Anweisung `#[no_mangle]` stehen, damit die Funktion aus einem C Kontext heraus aufrufbar ist. Außerdem sollte die *octolib* Bibliothek mit der Anweisung `extern crate octolib;` eingebunden werden. Eine minimale Haupt-Quelldatei eines invasiven *Cargo*-Projekts wird in Listing 3.3 veranschaulicht.

3.2.2 Kompilieren

Vor der Kompilierung wird im Falle, dass *leon* als Zielarchitektur ausgewählt wurde, zuerst eine wie in Kapitel 3.1 erläuterte JSON-Spezifikationsdatei generiert, in der der Pfad zum SPARC-V8 `gcc` Linker automatisch eingetragen wird. Für die anderen beiden Zielarchitekturen ist ein solcher Schritt nicht notwendig, da diese offiziell von `rustc` und `cargo` unterstützt werden. Nachdem diese Spezifikationsdatei generiert wurde, wird das Projekt mithilfe von `cargo` als statische Bibliothek kompiliert. Wur-

de das Projekt erfolgreich kompiliert, wird anschließend ein minimales C-Programm generiert, welches die von *OctoPOS/iRTSS* benötigte **main_ilet**-Funktion implementiert und innerhalb dieser die **rust_main_ilet**-Funktion aufruft. Im Anschluss daran wird dann noch die **shutdown** Funktion aufgerufen, um die Ausführung des Programms zu beenden. Die generierte C-Datei wird mithilfe eines passenden **gcc**-Compilers als Objekt-Datei kompiliert und anschließend mit *OctoPOS/iRTSS* und der zuvor kompilierten statischen Rust-Bibliothek verlinkt. Nun sollte eine ausführbare Datei existieren, welche auf einem Rechner mit x86-Architektur im Falle der „x86guest“ Architektur nativ ausführbar ist oder im Falle von *x64native* und *leon* mithilfe eines Emulators wie **qemu**.

Im Anschluss an die Kompilierung werden, vorausgesetzt die **--keep**-Option wurde nicht verwendet, jegliche temporäre Dateien gelöscht, beispielsweise die statische Rust-Bibliothek, die minimale C Datei oder auch die C Objektdatei.

3.3 *octolib*

Zusätzlich zum *octorust*-Programm wurde ebenfalls eine Rust-Bibliothek namens *octolib* geschrieben. Diese Bibliothek soll die Interaktion zwischen Rust und *OctoPOS/iRTSS* ermöglichen und an die neue Programmiersprache anpassen. *OctoPOS/IRTSS* bieten eine C-Schnittstelle an, welche man mithilfe von Rusts *Foreign Function Interface* und der *libc* Bibliothek aus Rust heraus ansprechen kann. Die *libc*-Bibliothek bietet die Möglichkeit, Komponenten, welche die Standardbibliothek benötigen, auszulassen, daher ist sie auch auf der SPARC-V8 Architektur nutzbar.

Die *octolib*-Bibliothek wird vor dem Kompilieren als Abhängigkeit in die **Cargo.toml**-Datei des zu kompilierenden *Cargo*-Projekts eingefügt. Hierfür wird mithilfe des **toml** Python Moduls der derzeitige Inhalt der **Cargo.toml**-Datei eingelesen, der Pfad zur lokal installierten *octolib*-Bibliothek in diese Daten injiziert und anschließend wieder in die Konfigurationsdatei geschrieben. Um zu vermeiden, dass die **Cargo.toml**-Datei Werte enthält, die auf eine lokale Konfiguration basieren, wird zunächst eine Kopie der ursprünglichen Daten im Hauptspeicher behalten. Nachdem die Kompilierung beendet wurde, werden diese Daten wieder in die **Cargo.toml**-Datei geschrieben. Damit die Bibliothek für jedes zu kompilierende Programm auffindbar ist, wird eine lokale Kopie während der Installation des *octorust*-Programms im **~/.octorust** Verzeichnis erstellt.

3.3.1 Struktur

Die *octolib* Bibliothek besteht aus einer Hauptbibliotheksdatei namens `lib.rs`, welche alle anderen Module innerhalb des Projekts einbindet. Diese Datei implementiert außerdem die in Kapitel 3.1 erwähnten Funktionen `eh_personality`, `eh_unwind_resume` und `panic_fmt`, sodass nicht jedes einzelne Projekt diese aufs Neue implementieren muss. Sobald die *octolib* Bibliothek mit der Anweisung `extern crate octolib;` ins Projekt eingebunden wurde, werden diese Funktionen ebenfalls importiert.

Die Bibliothek bietet die folgenden Module:

bindings Dieses Modul enthält die direkten C-Rust *Bindings* zur C-Schnittstelle von *OctoPOS/iRTSS*.

helper Dieses Modul enthält eine Sammlung an Hilfsfunktionen, die nicht direkt mit dem *OctoPOS* Betriebssystem oder der *iRTSS* Middleware in Verbindung stehen, aber trotzdem bei der Programmierung nützlich sein können.

improvements Dieses Modul enthält Rust-spezifische Abstraktionen, die es dem Programmierer erleichtern, Rust-Programme für invasive Systeme zu schreiben.

octo_structs Dieses Modul enthält Rust-äquivalente Strukturen zu denen, die in der C-Schnittstelle verwendet werden.

octo_types Dieses Modul enthält Rust-äquivalente Typen zu denen, die in der C-Schnittstelle verwendet werden.

3.3.2 Direkte C-Rust-*Bindings*

Im ersten Schritt werden die Funktionen der C-Schnittstelle direkt Eins-zu-Eins als Rust-Funktionen eingebunden. Dank des *Foreign Function Interface* müssen lediglich die Parameter- und Rückgabetypen an die neue Programmiersprache angepasst werden. Für die meisten Typen gibt es direkte Äquivalente in Rust, jegliche anderen Typen stellt die *libc*-Bibliothek zur Verfügung. Ein nennenswertes Beispiel eines Typs, der nicht in Rust enthalten ist, ist der `void` Typ. Dieser Typ kann durch den `c_void` Typen aus der *libc*-Bibliothek emuliert werden. `void`-Zeiger (`void*`), welche an einigen Stellen der C-Schnittstelle verwendet werden, können so als `*mut c_void` in Rust dargestellt werden.

Um die Funktionen erfolgreich in die Rust-Bibliothek einbinden zu können, muss

man einen **extern**-Block erstellen und dort die Funktionsdefinitionen vornehmen. Damit diese auch von anderen Modulen oder Projekten aufgerufen werden können, müssen diese als **pub fn** deklariert werden. Da diese Funktionen lediglich importierte C-Funktionen ohne jegliche Sicherheitsgarantien sind, muss man bei jedem Gebrauch dieser Funktionen einen **unsafe**-Block verwenden.

Strukturen und Typen

Zusätzlich zu den Funktionsdefinitionen mussten die in der C-Schnittstelle definierten Strukturen und Typen ebenfalls portiert werden. Dank der **#[repr(C)]** Anweisung kann man in Rust Strukturen erstellen, welche kompatibel mit ihren C-Äquivalenten sind. Diese Strukturen sind teils von Plattform-spezifischen Konstanten abhängig, welche man in Rust mit der **#[cfg(target_arch = "Zielarchitektur")]**-Anweisung für unterschiedliche Architekturen definieren kann. Benutzerdefinierte Typen, welche prinzipiell Aliase für bestehende Typen sind, lassen sich in Rust durch das Schlüsselwort **type** erstellen.

Das `helper`-Modul

Das **helper**-Modul enthält Adapter für C-Funktionen, für die in Rust ohne Zugriff zur Standardbibliothek keine Alternative existieren. So werden beispielsweise mehrere Adapter um die **printf**-Funktion aus der C-Standardbibliothek geboten, die je eine andere Anzahl an zusätzlichen Parametern bieten, um Variablen ausgeben zu können.

Beachtet werden muss bei diesen print-Funktionen, dass die überreichten Zeichenketten manuell null-terminiert werden müssen. Um „Hello World“ auszugeben, muss man somit die Anweisung **print("Hello World\n\0");** verwenden.

3.3.3 Rust-spezifische Verbesserungen

Im Folgenden werden verschiedene Abstraktionen über die direkten C-Rust *Bindings* erstellt, welche das Programmieren vereinfachen und die Stärken von Rust ausnutzen sollen.

Constraints

Die `Constraints`-Struktur ist eine objektorientierte Abstraktion für die *Constraints*, mit denen die Anzahl Ressourcen die einem *Claim* zur Verfügung stehen sollen, spezifiziert werden. Die Implementierung dieser Struktur bietet einen Konstruktor, welcher eine neue `constraints_t`-Struktur mithilfe der `agent_constr_create`-Funktion aus dem `octo_bindings`-Modul erstellt und anschließend einige Standardwerte setzt. Dem Konstruktor werden zudem 2 Parameter übergeben, welche die minimale und maximale Anzahl an Rechenelementen spezifizieren.

Alle Parameter der *Constraints* lassen sich nachträglich durch Methoden der Struktur ändern. Hierbei profitieren vor allem die Methoden, welche boolesche Werte als Parametertypen besitzen, denn diese werden in den direkten C-Rust-*Bindings* statt mit `bool`-Werten mit `u8`-Werten aufgerufen. Dies liegt daran, dass es in C keinen `bool` oder ähnlichen Typen gibt. Dort werden stattdessen Zahlenwerte auf den Wert `0` geprüft. Dies wird in den Methoden dieser Struktur jedoch abstrahiert, so dass man in Rust den `bool`-Typ verwenden kann.

Außerdem unterstützt die Methode `merge_constraints` anstelle einer `constraints_t`-Struktur aus der C-Schnittstelle eine in Rust definierte `Constraints`-Struktur. Möchte man trotzdem lieber eine `constraints_t`-Struktur verwenden, so kann stattdessen die `merge_constraints_t`-Methode verwendet werden.

Um Zugriff auf die interne `constraints_t`-Struktur zu erhalten, kann die `to_constraints_t`-Methode aufgerufen werden. Hiernach ist die `Constraints`-Struktur selbst nicht mehr verwendbar, da die interne `constraints_t`-Variable aufgrund der *Move-Semantik* nicht mehr der *Owner* des zugehörigen Speicherbereichs ist.

Closures

Rust Closures können nicht ohne Weiteres einer C-Schnittstelle als Parameter übergeben werden. Somit ist es mit den direkten C-Rust-*Bindings* nicht möglich, Closures zu verwenden, um Code auf den Rechenelementen ausführen zu lassen. Dies wäre jedoch eine wünschenswerte Funktionalität. Da Closures aus der Funktion selbst, als auch aus ihrem Erstellungskontext bestehen [10], ist es nicht möglich, eine Closure direkt als eine normale Rust-Funktion zu verwenden. Stattdessen erstellt man einen Zeiger auf die Datenregion, welche die Closure repräsentiert und übergibt diesen daraufhin einer `extern "C"` Funktion, welche aus diesem Zeiger die Closure in einem `unsafe`-Block zurück konvertiert. Diese `extern "C"`-Funktion kann dann der C-Schnittstelle

zusammen mit dem Zeiger auf die Closure-Daten übergeben werden.

Die Erstellung des Closure-Zeigers wird direkt in der später genauer erläuterten `infect`-Methode der `AgentClaim`-Struktur erledigt, während zur Rückkonvertierung eine `extern "C"`-Funktion namens `execute_closure` erstellt wurde.

AgentClaim

Das wichtigste Konstrukt in der *octolib* Bibliothek ist die `AgentClaim`-Struktur, da sie Abstraktionen für die *Invade*-, *Infect*- und *Retreat*-Phasen implementiert. Diese Struktur bietet zunächst einmal eine Abstraktionsschicht über die `agentclaim_t`-Struktur aus der C-Schnittstelle und verschiedene Methoden um diese zu verwenden.

Der Konstruktor nimmt als einzigen Parameter eine `Constraints`-Struktur entgegen, welche verwendet wird, um die Ressourcen des *Claims* zu spezifizieren. Mit diesen *Constraints* wird dann eine interne `agentclaim_t`-Struktur initialisiert, welche für alle folgenden Methodenaufrufe verwendet wird. Es wird somit bereits im Konstruktor implizit die *Invade*-Phase des invasiven Rechnens ausgeführt.

Eine praktische Methode für das *Debugging* ist `set_verbose`. Diese Methode erlaubt es, die Ausführlichkeit der auf die Kommandozeile ausgegebenen Informationen der `AgentClaim`-Struktur zu beeinflussen.

Die `reinvade`-Methode erlaubt es, eine *Reinvade*-Operation auf dem *Claim* auszuführen. Diese erlaubt es, die von der internen `agentclaim_t`-Struktur verwendeten *Constraints* anzupassen. Diese Methode nimmt einen optionalen Parameter an, mit dem man neue *Constraints* für den *Claim* spezifizieren kann. Werden neue *Constraints* gesetzt, werden die alten *Constraints* mit der `agent_constr_delete`-Funktion aus dem Speicher gelöscht.

Die wohl wichtigste Methode der `AgentClaim`-Struktur ist die `infect`-Methode. Mit dieser kann eine invasive *Infect*-Operation auf dem *Claim* ausgeführt werden. Dieser Methode wird eine Funktion oder eine Closure als Parameter übergeben, welche dann auf den Rechenelementen des *Claims* ausgeführt wird. Zusätzlich können ebenfalls Parameterdaten für die einzelnen Recheneinheiten übergeben werden. Diese Parameterdaten werden als ein Array von `void`-Zeigern übergeben. Die Anzahl der Elemente des Arrays muss mit der Anzahl der vom *Claim* reservierten Rechenelemente übereinstimmen. Bei einer zu geringen Anzahl an Rechenelementen kann es ansonsten zu Fehlern kommen. Bei einer zu großen Zahl an Rechenelementen kommt es zum Pufferüberlauf, was den Absturz des Programms zur Folge hat. Um die Parameterübergabe sicherer zu gestalten, können in der Zukunft Serialisierungsbibliotheken wie `serde` verwendet werden.

In der `infect`-Methode werden Signal-Strukturen aus der C-Schnittstelle verwendet, um die Kommunikation zwischen verschiedenen Rechenelementen zu verwalten.

Um dies zu erreichen, wird die übergebene Funktion oder Closure nochmals von einer internen Closure umgeben, welche eine Referenz auf eine in der `infect`-Methode erstellten Signal-Struktur besitzt. Über dieses Signal wird signalisiert, wann die übergebene Funktion oder Closure auf einem Rechenelement erfolgreich ausgeführt wurde. Hierdurch ist es möglich, nach der Initialisierung der *i-lets* auf die vollständige Abarbeitung dieser zu warten, oder alternativ das Signal, mithilfe dessen die *i-lets* mit dem Hauptfaden des Programms kommunizieren, als Rückgabewert zu verwenden. Um beide dieser Vorgehensweisen anzubieten, gibt es zusätzlich zur `infect`-Methode die `infect_async`-Methode. Die `infect`-Methode wartet auf die vollständige Ausführung aller *i-lets*. Die `infect_async`-Methode gibt hingegen die `simple_signal`-Struktur, mit der die *i-lets* das Ende ihrer Ausführung signalisieren, als Rückgabewert zurück. Anschließend kann der Nutzer mit der `simple_signal_wait`-Funktion aus dem `octo_bindings` Modul auf die vollständige Ausführung aller *i-lets* warten. Die interne Closure, welche die Kommunikation über Signale implementiert, wird innerhalb der `infect` Methode zu einem `void`-Zeiger konvertiert, welcher den *i-lets* anschließend als Datenparameter übergeben wird.

Für jedes Rechenelement wird iteriert und ein *i-let* initialisiert. Diese *i-lets* erhalten die `execute_closure`-Funktion als ihren Funktionsparameter und den Closure-Zeiger als ihren ersten Datenparameter. Wurden der `infect`-Methode zusätzliche Datenparameter übergeben, so werden diese als zweite Datenparameter den *i-lets* übergeben. Nachdem ein i-let initialisiert wurde, wird die eigentliche *Infect*-Operation durchgeführt.

Das bereits im Grundlagenkapitel 2.1.2 erläuterte `Drop`-Trait für Rust-Strukturen wird für die `AgentClaim`-Struktur implementiert. Verlässt eine Instanz der `AgentClaim`-Struktur den Geltungsbereich, wird eine `Retreat`-Operation auf der internen `agentclaim_t`-Struktur ausgeführt und zusätzlich die *Constraints* des *Claims* gelöscht. Dieses implizite *Retreat* stellt sicher, dass die vom Claim verwendeten Ressourcen nach dem Gebrauch wieder freigegeben werden und somit anderen Programmteilen zur Verfügung stehen.

4 Evaluation

Im Folgenden wird der Gebrauch von Rust im Zusammenhang mit dem invasiven Rechnen evaluiert. Hierbei wird vor allem mit den bereits unterstützten Sprachen C und X10 verglichen.

Jedes der folgenden Programme wurde auf der folgenden Hardware/Software Konfiguration kompiliert und ausgeführt:

CPU	Intel Core i5-5200U @ 2.2GHZ x 2
Hauptspeicher	8GB
Betriebssystem	Ubuntu 16.04.3 LTS
Linux Kernel	Version 4.4.0-97-generic
gcc(x86)	5.4.0
gcc(SPARC-V8)	7.2.0
iRTSS	2017-06-07-nightly, x86guest-generic
rustc	1.19.0-nightly
JDK	openjdk 1.8.0_131
octorust	Version 1.0.0
x10i	Commit 31183335a89917f489046da746c5181174a7bdb3

Tabelle 4.1: Dies ist die Hard- und Software Konfiguration des Rechners, auf dem die nachfolgenden Programme kompiliert und ausgeführt wurden.

Für jedes betrachtete Programm wurden je zwei Versionen pro Programmiersprache betrachtet. Zum einen mit Compiler-Optimierungen, zum anderen ohne. Bei Messungen der Ausführungszeit wurde die Wall-Clock-Time in Sekunden als Messwert verwendet. Jedes Programm, dessen Laufzeit überprüft wird, wurde mithilfe des `temci`-Benchmarkprogramms fünfzig mal ausgeführt. C- und Rust-Programme wurden mit *octorust* kompiliert, X10-Programme mit dem *x10i* Compiler. Alle berechneten Werte wurden auf vier Nachkommastellen gerundet.

Um die statistische Relevanz der Ergebnisse zu evaluieren, wurden zu jeder Messung der Laufzeit das arithmetische Mittel (Abbildung 4.1), der Median (Abbildung 4.2), die Stichproben-Standardabweichung (Abbildung 4.3) und der zugehörige Stichproben-Variationskoeffizient (Abbildung 4.4) berechnet [28].

Abbildung 4.1: Das Arithmetische Mittel

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Abbildung 4.2: Der Median

$$\tilde{x} = \begin{cases} x_{\frac{n+1}{2}} & n \text{ ungerade} \\ \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}) & n \text{ gerade} \end{cases}$$

Abbildung 4.3: Die Stichproben-Standardabweichung

$$s_x = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Abbildung 4.4: Der Stichproben-Variationskoeffizient für Werte $x_i > 0$

$$v_x = \frac{s_x}{\bar{x}}$$

4.1 Kompilierungsdauer und Dateigröße

Zu Beginn werden die Kompilierungsdauer und die Dateigröße bei Gebrauch der einzelnen Programmiersprachen verglichen. Diese Werte wurden für ein simples Programm betrachtet, welches lediglich die Ausführung startet und anschließend sofort wieder beendet.

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	0.4924	0.4900	0.0170	3.4465%
C (opt)	0.5586	0.5600	0.0134	2.3993%
Rust	0.8174	0.8200	0.0164	2.0051%
Rust (opt)	0.8116	0.8100	0.0136	1.6765%
X10	82.8640	83.0750	1.3321	1.6076%
X10 (opt)	192.1182	191.9150	1.4026	0.7301%

Tabelle 4.2: Diese Tabelle stellt die Messergebnisse der Kompilierungsdauer des *startup*-Programms dar. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

Sprache	Dateigröße
C	10117708
C (opt)	10117708
Rust	10118632
Rust (opt)	10117760
X10	16630348
X10 (opt)	17546444

Tabelle 4.3: Diese Tabelle stellt die Messergebnisse der Dateigrößen des *startup*-Programms dar. Die Messwerte sind in der Anzahl Bytes, welche die ausführbare Datei ausmachen, angegeben.

Die Ergebnisse in Tabelle 4.2 zeigen, dass X10-Programme im Vergleich zu C- oder Rust-Programmen eine ungefähr 100-fache Kompilierungsdauer vorweisen. Mit aktivierten Compiler-Optimierungen steigt diese Zahl auf über das 200-fache. Zudem sind die Dateigrößen des X10-Programme mindestens 6511716 Byte größer als die der C- und Rust-Programme.

4.2 Laufzeitverhalten

Um die erreichbare Leistung der Programmiersprachen zu vergleichen, wurden im Folgenden unterschiedliche Programme entwickelt. Diese Programme wurden soweit möglich in allen Programmiersprachen einheitlich implementiert und ermöglichen so, das Laufzeitverhalten der Programmiersprachen miteinander zu vergleichen.

4.2.1 Vergleich der Anlaufzeit

Es wird zunächst überprüft, wie lange ein Programm, welches in einer zu betrachtenden Programmiersprachen geschrieben wurde, benötigt, um die Ausführung zu starten und anschließend sofort wieder zu beenden. Es wurde dasselbe Programm verwendet, welches in Kapitel 4.1 zur Messung der Kompilierungsdauer und Dateigröße verwendet wurde.

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	0.3772	0.3700	0.0140	3.7123%
C (opt)	0.3852	0.3800	0.0171	4.4268%
Rust	0.3820	0.3800	0.0139	3.6258%
Rust (opt)	0.4026	0.3800	0.0459	11.4080%
X10	1.7738	1.7800	0.0255	1.4357%
X10 (opt)	1.7506	1.7500	0.0234	1.3380%

Tabelle 4.4: Diese Tabelle stellt die Messergebnisse der Laufzeit des *startup*-Benchmarks dar. Dieser Benchmark simuliert das Starten und sofortige Schließen eines Programms. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

Anhand der Werte in Tabelle 4.4 kann man prinzipiell erkennen, dass die Anlaufzeiten von C und Rust sich sehr nahe sind, etwaige Unterschiede lassen sich auf die Varianz der Ergebnisse zurückführen. Die Implementierung in X10 benötigt jedoch im Schnitt ungefähr 1,3-1,4 Sekunden mehr Zeit als es bei den beiden anderen Programmiersprachen der Fall ist.

4.2.2 Berechnen von Primzahlen

Um die Rechenleistung der verschiedenen Programmiersprachen bei einem intensiveren mathematischen Problem zu vergleichen, wurden Programme geschrieben, welche Primzahlen berechnen. Hierbei wurden zwei unterschiedliche Ansätze verwendet. Zum einen wurde eine naive Berechnung implementiert, welche jede Zahl individuell auf Teilbarkeit mit kleineren Zahlen prüft. Andererseits wurde das Sieb von Eratosthenes verwendet, welches eine effiziente Methode zum Berechnen von Primzahlen ist.

Im Falle des Siebs von Eratosthenes kann Rust in diesem Kontext keine Quadratwurzeloperation durchführen, da diese die Standardbibliothek benötigt. Daher wurden in allen Sprachen anstelle einer Quadratwurzelfunktion feste Zahlen verwendet, um den Vergleich zwischen den Sprachen gerecht zu gestalten. Da jedoch nur eine einzelne Quadratwurzeloperation für das Sieb des Eratosthenes benötigt wird, wäre dies ohnehin aller Wahrscheinlichkeit nach kein entscheidender Faktor bei der Laufzeitbewertung.

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	4.6898	4.6950	0.0313	0.6681%
C (opt)	4.2134	4.2150	0.0297	0.7041%
Rust	61.6462	61.6050	0.2849	0.4621%
Rust (opt)	4.9692	4.9650	0.0319	0.6413%
X10	10.5830	10.5500	0.1368	1.2923%
X10 (opt)	5.9694	5.9700	0.0719	1.2039%

Tabelle 4.5: Diese Tabelle stellt die Ergebnisse des *naive-primes*-Benchmarks dar. Dieser Benchmark überprüft die ersten 50000 natürlichen Zahlen darauf, ob es sich bei ihnen um Primzahlen handelt, indem er für jede Zahl individuell alle kleinere Zahlen auf Teilbarkeit prüft. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

Mit den Werten in den Tabellen 4.5 und 4.6 lässt sich keine eindeutige Aussage darüber treffen, welche der Programmiersprachen die Aufgabe am effizientesten gelöst hat, vorausgesetzt, es werden die Versionen mit Compiler-Optimierungen betrachtet. Im Schnitt weisen C und Rust jedoch eine 1-1,8 Sekunden bessere Laufzeit als X10 auf. Allerdings muss dabei auch die längere Anlaufzeit von X10 Programmen in Betracht gezogen werden. Zieht man diese ab, sind die Ergebnisse zu nah aneinander, als dass man eine klare Tendenz beobachten könne.

Werden allerdings die Versionen ohne Compiler-Optimierungen betrachtet, gibt es deutliche Unterschiede zwischen den Sprachen. In Tabelle 4.5 ist beispielsweise

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	0.6250	0.6150	0.0196	3.1382%
C (opt)	0.5584	0.5500	0.0175	3.1408%
Rust	0.8362	0.8300	0.0138	1.6544%
Rust (opt)	0.4984	0.4900	0.0210	4.2196%
X10	2.3508	2.3400	0.0347	1.4782%
X10 (opt)	1.9892	1.9900	0.0274	1.3771%

Tabelle 4.6: Diese Tabelle stellt die Ergebnisse des *eratosthenes-primes*-Benchmarks dar. Dieser Benchmark überprüft mithilfe des Siebs von Eratosthenes die ersten 1690000 natürlichen Zahlen darauf, ob es sich bei ihnen um Primzahlen handelt. Da Rust ohne die Standardbibliothek keine Quadratwurzeloperationen unterstützt, wurde in jeder Implementierung der Wert 1300 fest als das Ergebnis der Quadratwurzeloperation einprogrammiert. Die gegebenen Werte wurden gewählt, da höhere Werte in Rust und C einen Stack-Überlauf zur Folge hätten. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

zu erkennen, dass die unoptimierte Version des Rust-Programms ungefähr 6-mal langsamer als die nächstlangsamste betrachtete Konfiguration ist.

4.2.3 Paralleles Berechnen von Primzahlen

Um auch die invasiven Aspekte der Sprachen zu evaluieren, wurden X10-, C- und Rust-Programme geschrieben, welche Primzahlen in einer parallelen Art und Weise berechnen. Hierfür wird die Menge an Zahlen, welche überprüft werden sollen, ob es sich bei ihnen um Primzahlen handelt, gleichermaßen in Partitionen aufgeteilt. Die Anzahl und Größe der Partitionen hängen hierbei von der Anzahl an verfügbaren Rechenelementen ab. Jedes der Rechenelemente berechnet anschließend die ihm zugewiesene Menge an Zahlen naiv, indem jede Zahl mit jeder kleineren Zahl auf Teilbarkeit geprüft wird.

Der Vergleich der Werte in Tabelle 4.7 und 4.8 lässt, anders wie in Kapitel 4.2.2, Aussagen zur Laufzeiteffizienz der einzelnen Sprachen zu. Vor allem in Tabelle 4.7 kann man erkennen, dass C mit 30 Sekunden die beste Laufzeit aufweist, gefolgt von X10 mit 31 Sekunden. Die schlechteste Laufzeit wies Rust mit 38 Sekunden auf.

Zur Bewertung von parallelen Algorithmen verwendet man unter Anderem die Werte des *Speedup* und der *Effizienz*. Den relativen *Speedup* eines parallelen Programms

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	33.9812	33.7450	0.4918	1.4473%
C (opt)	29.9782	29.9750	0.0377	0.1257%
Rust	476.9242	475.8950	3.6647	0.7684%
Rust (opt)	37.6212	37.3300	0.6651	1.7679%
X10	67.9618	67.4050	1.1505	1.6929%
X10 (opt)	31.4584	31.3050	0.4181	1.3290%

Tabelle 4.7: Diese Tabelle stellt die Laufzeitergebnisse des sequentiellen Ergebnisses des *parallel-primes*-Benchmarks dar, es wurde also die zu verwendende Anzahl an Rechenelementen auf `1` gesetzt. Dieser Benchmark überprüft die ersten `500000` natürlichen Zahlen darauf, ob es sich bei ihnen um Primzahlen handelt. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	16.4720	16.4600	0.0528	0.3203%
C (opt)	13.8018	13.7950	0.0352	0.2551%
Rust	242.6560	242.4800	0.7602	0.3133%
Rust (opt)	17.8830	17.8700	0.0862	0.4823%
X10	46.3264	46.3800	0.2348	0.5069%
X10 (opt)	16.8028	16.7600	0.1181	0.7026%

Tabelle 4.8: Diese Tabelle stellt die Laufzeitergebnisse des *parallel-primes*-Benchmarks dar, wenn `8` Rechenelemente verwendet werden. Dieser Benchmark überprüft die ersten `500000` natürlichen Zahlen darauf, ob es sich bei ihnen um Primzahlen handelt. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

Abbildung 4.5: Der Speedup eines parallelen Algorithmus mit n Rechenelementen

$$S(n) = \frac{T(1)}{T(n)}$$

Abbildung 4.6: Die Effizienz eines parallelen Algorithmus mit n Rechenelementen

$$E(n) = \frac{S(n)}{n}$$

berechnet man mit der Formel aus Abbildung 4.5 und die *Effizienz* mit der Formel aus Abbildung 4.6. $S(n)$ bezeichnet hierbei den *Speedup*, $E(n)$ die *Effizienz* und $T(x)$ die Ausführungszeit mit x Prozessoren.

Mit den Werten aus Tabelle 4.7 und 4.8 und diesen Formeln kann man nun auch den Speedup und die Effizienz für das implementierte Programm für die unterschiedlichen Varianten berechnen.

Sprache	$S_{\bar{x}}(8)$	$S_{\tilde{x}}(8)$	$E_{\bar{x}}(8)$	$E_{\tilde{x}}(8)$
C	2.0630	2.0501	0.2579	0.2563
C (opt)	2.1721	2.1729	0.2715	0.2716
Rust	1.9654	1.9626	0.2457	0.2453
Rust (opt)	2.1037	2.0890	0.2630	0.2611
X10	1.4670	1.4533	0.1834	0.1817
X10 (opt)	1.8722	1.8678	0.2340	0.2335

Tabelle 4.9: In dieser Tabelle werden die berechneten Werte für den *Speedup* und die *Effizienz* des *parallel-primes-Benchmark* für acht Recheneinheiten veranschaulicht. Hierbei wurden die Ergebnisse sowohl für den Median als auch für das arithmetischen Mittel berechnet. Alle Werte sind auf vier Nachkommastellen gerundet.

Betrachtet man die Ergebnisse der *Speedup*- und *Effizienz*berechnung in Tabelle 4.9, so kann man keine klare Tendenz zwischen Rust und C feststellen. Es kann also diesbezüglich nicht behauptet werden, dass eine der Sprachen durch die Parallelisierung stärker profitiert. Die X10-Programme weisen jedoch in jedem Fall einen um mindestens 0,2 geringeren Speedup auf und auch die Effizienz ist mindestens um 0,3 geringer als die Ergebnisse der anderen Programmiersprachen. Es kann also behauptet werden, dass Rust und C in diesem Fall stärker von der Parallelisierung profitieren als X10

Der *Speedup* ist mit einem Wert von rund 1,5 bis 2,2 in diesem Beispiel eher gering, dies liegt jedoch am verwendeten Algorithmus, der durch die gleichmäßige Partitionierung der betrachteten Zahlen und der naiven Berechnung Leistungseinbußen in Kauf nimmt. Dies liegt vor allem daran, dass bei der naiven Berechnung größere Zahlen einen höheren Rechenaufwand erfordern. Da jedoch alle Partitionen gleich groß sind, wird der *i-let* mit den höchsten Werten länger als alle anderen *i-lets* benötigen, um seine Ausführung zu beenden.

4.2.4 Allokation auf dem Heap

X10 verwaltet den Speicher mithilfe eines *Garbage Collectors*, wobei C und Rust auf einen solchen verzichten. Während *Garbage Collector* ein sehr hilfreiches Werkzeug sind, um den Programmieraufwand zu verringern, so kann dies allerdings auch auf Kosten der Laufzeiteffizienz und Verfügbarkeit durch *Garbage Collector*-Pausen geschehen.

Um zu Prüfen, ob diese Gegebenheit einen messbaren Effekt auf die Laufzeit eines Programmes haben kann, wurde ein Programm geschrieben, welches kontinuierlich Objekte auf dem Heap erstellt, welche anschließend wieder aus dem Geltungsbereich verschwinden. In C muss der Speicher, in dem diese Objekte gespeichert werden, manuell mit `free` befreit werden, in Rust wird dieser Speicher automatisch wieder befreit sobald ihr *Owner* den Geltungsbereich verlässt und im Falle von X10 werden diese Objekte vom *Garbage Collector* verwaltet.

Die erstellten Objekte sind in diesem Fall Arrays von Integer-Zahlenwerten. In C und X10 wurden die `int`-Typen, in Rust der `i32` Typ verwendet. Das Äquivalent zu Arrays in X10 sind Rails, daher wurden in der X10 Implementierung diese verwendet.

Wie man den Ergebnissen der Tabelle 4.10 entnehmen kann, benötigt X10 im Vergleich zu Rust oder C ein Vielfaches an Ausführungszeit, um die selbe Anzahl an gleich großen Objekten zu erstellen. Betrachtet man die Ergebnisse der unoptimierten Versionen der Programme, benötigt X10 mehr als 11-mal so lange wie Rust und ungefähr 17-mal so lange wie C. Dies könnte eine Konsequenz des *Garbage Collectors* sein, es lässt sich jedoch ohne eine genaue Messung des Speicherverhaltens keine eindeutige Aussage diesbezüglich treffen.

Bei häufiger Erstellung von neuen Objekten weisen Rust und C durch die gemessene Laufzeit einen nennenswerten Vorteil gegenüber X10 auf. Im Falle von Rust muss der Programmierer obendrein den Speicher nicht selbst wieder freigeben, denn dies geschieht implizit sobald eine Variable den Geltungsbereich verlässt. Somit weist Rust gegenüber C ebenfalls trotz der ungefähr 10 Sekunden schlechteren Laufzeit einen Vorteil auf.

Sprache	$\bar{x}$	$\tilde{x}$	s_x	v_x
C	19.0216	19.0000	0.0743	0.3904%
C (opt)	0.3854	0.3700	0.0206	5.3513%
Rust	29.5478	29.4800	0.2671	0.9038%
Rust (opt)	5.1024	5.1000	0.0258	0.5048%
X10	339.3450	338.1900	2.3484	0.6920%
X10 (opt)	227.8816	226.0900	2.8234	1.2390%

Tabelle 4.10: Diese Tabelle stellt die Ergebnisse des *garbageonly*-Benchmarks dar. Dieser Benchmark erstellt insgesamt `1000000` `int`/`i32`-Arrays/Rails mit einer Größe von `5000` Elementen, welche alle auf den Wert `0` initialisiert werden. Alle Werte sind auf vier Nachkommastellen gerundet und mit Ausnahme des Variationskoeffizienten in Sekunden angegeben.

4.3 Aufwand des Programmierens

Im Folgenden wird der Aufwand des Erstellens eines invasiven Programms in jeder der betrachteten Programmiersprachen verglichen.

4.3.1 Projektstruktur

Zunächst wird die Struktur eines invasiven Projekts in den unterschiedlichen Programmiersprachen verglichen.

Einzelne C- und Rust-Programme können jeweils mithilfe von *octorust* kompiliert werden. Gleiches gilt für X10, für welches *x10i* die Kompilierung durchführt.

Zusätzlich hierzu bietet *octorust* jedoch auch das Kompilieren von *Cargo*-Projekten. Diese weisen zwar einen etwas höheren Arbeitsaufwand zu Beginn des Projekts auf, erlauben aber anschließend benutzerfreundliches Einbinden von externen Bibliotheken in der `Cargo.toml`-Konfigurationsdatei. Außerdem wird in *Cargo*-Projekten die *octolib*-Bibliothek automatisch eingebunden. Im Falle von X10 und C ist eine ähnliche Funktionalität nicht vorhanden, externe Bibliotheken müssen beim Kompilieren manuell eingebunden werden.

4.3.2 Minimales Invade, Infect, Retreat

Um den Programmieraufwand eines simplen invasiven Programms zu beurteilen, wurde ein Programm geschrieben, welches zuerst eine *Invade*-Operation durchführt und anschließend eine simple „Hello World“-Funktion auf jedem der reservierten Rechenelemente in der *Infect*-Phase ausführt. Das Programm wartet anschließend bis alle Rechenelemente mit der Ausführung beendet sind und gibt daraufhin alle Ressourcen in der *Retreat*-Phase wieder frei.

Rust

Um dieses Programm in Rust zu implementieren, muss man zunächst die notwendigen Strukturen importieren. Benötigt werden die `Constraints`- und `AgentClaim`-Strukturen. Zusätzlich muss der `libc::c_void`-Typ importiert werden, um die Funktionssignatur der *i-let* Funktionen zu definieren. Außerdem wird zur Ausgabe von „Hello World!“ die `helper::printer::print`-Funktion benötigt.

Zunächst werden die *Constraints* mithilfe der `Constraints`-Struktur initialisiert. Anschließend können diese weiter konfiguriert werden oder direkt bei der Initialisierung einer Instanz der `AgentClaim`-Struktur verwendet werden. Wird der Konstruktor der `AgentClaim`-Struktur aufgerufen, so wird die *Invade*-Operation implizit durchgeführt. Es muss nun nur noch die `infect`-Methode aufgerufen werden. Dieser Methode wird eine Closure als Parameter übergeben, welche den Text „Hello World“ auf die Kommandozeile ausgibt. Sobald die `infect`-Methode aufgerufen wurde, wartet der Hauptfaden des Programms darauf, dass die einzelnen Rechenelemente ihre Ausführung beendet haben. Hiernach muss der Programmierer keine zusätzlichen Instruktionen aufrufen. Die *Retreat*-Phase wird beim Verlassen des Geltungsbereichs automatisch durchgeführt.

Eine beispielhafte Implementierung dieses Programms wird in Listing 4.1 veranschaulicht.

X10

Soll das Programm in X10 implementiert werden, so muss man zuerst die benötigten Klassen importieren. Es werden mindestens die Klassen
`invasic.constraints.PEQuantity`, `invasic.IncarnationID` und
`invasic.Claim` benötigt. Zusätzlich muss `x10.io.Console` importiert werden, um die Ausgabe auf die Kommandozeile zu ermöglichen.

Listing 4.1: Minimales Invade, Infect, Retreat in Rust

```
#![no_std]

extern crate libc;
extern crate octolib;
use libc::c_void;
use octolib::helper::printer::print;
use octolib::improvements::constraints::Constraints;
use octolib::improvements::claim::AgentClaim;

#[no_mangle]
pub extern "C" fn rust_main_ilet(claim_id: u8) {

    let mut ilet_fn = |params: *mut c_void| {
        print("Hello World\n\0");
    };

    let mut constr = Constraints::new(4, 4);
    let mut claim = AgentClaim::new(constr);
    claim.infect(ilet_fn, None);
}
```

Wie bereits bei Rust werden zunächst die *Constraints* spezifiziert. Hierzu ruft man den Konstruktor der Klasse **`PEQuantity`** auf. Man kann mithilfe des **`&&`**-Operators weitere *Constraints*-Attribute wie beispielsweise **`TileSharing.WITH_OTHER_APPLICATIONS`** mit der **`PEQuantity`** verbinden. Anschließend führt man die *Invade*-Phase mithilfe der statischen **`Claim.invade`**-Methode aus und erhält als Rückgabewert den erstellten *Claim*. Um daraufhin die *Infect*-Phase zu beginnen, muss auf diesem *Claim* die **`infect`**-Methode verwendet werden. Als Parameter erhält diese eine Closure, welche „Hello World“ auf die Kommandozeile ausgibt. Sobald die Methode aufgerufen wurde, wartet der Hauptfaden des Programms auf das Ende der Ausführung der verwendeten Rechenelemente. Nach erfolgreicher Ausführung der *Infect*-Phase muss dann noch manuell die **`retreat`**-Methode des *Claims* aufgerufen werden.

Eine beispielhafte Implementierung dieses Programms wird in Listing 4.2 veranschaulicht.

Listing 4.2: Minimales Invade, Infect, Retreat in X10

```
import x10.io.Console;

import invasic.constraints.PEQuantity;
import invasic.constraints.TileSharing;
import invasic.IncarnationID;
import invasic.Claim;

class Infect {
    public static def main(Array[String]) {
        val iletFn = (id: IncarnationID) => {
            Console.OUT.println("Hello World!");
        };

        val constraints = new PEQuantity(4, 4)
            && TileSharing.WITH_OTHER_APPLICATIONS;
        val claim = Claim.invade(constraints);
        claim.infect(iletFn);
        claim.retreat();
    }
};
```

C

Im Gegensatz zu Rust und X10 müssen die in C verwendeten Funktionen und Strukturen nicht alle explizit importiert werden. Es muss lediglich **`octopos.h`** mit der Präprozessor-Anweisung **`include`** verwendet werden. Um auf die Kommandozeile ausgeben zu können, muss zusätzlich **`stdio.h`** importiert werden.

Die Constraints werden mithilfe der **`agent_constr_create`**-Funktion initialisiert und müssen anschließend mit diversen **`agent_constr_set_...`**-Funktionen konfiguriert werden. Mit dieser konfigurierten **`constraints_t`**-Struktur kann nun eine **`agentclaim_t`**-Struktur mithilfe der **`agent_claim_invade`**-Funktion initialisiert werden.

Nach dieser *Invade*-Operation muss man in C manuell über alle reservierten Rechenelemente iterieren, dabei **`simple_ilet`**-Strukturen für diese erstellen und anschließend mit der **`proxy_infect`**-Funktion die Ausführung beginnen. Als *i-let*-Funktion muss eine C-Funktion verwendet werden, da die Sprache keine Closures oder ähnliche Konstrukte bietet. Damit der Hauptfaden auf das Ausführungsende der einzelnen Rechenelemente wartet, muss man manuell Signalstrukturen verwenden.

Nachdem die Recheneinheiten signalisiert haben, dass sie ihre Ausführung beendet

haben, muss man noch manuell die *Retreat*-Phase beginnen und anschließend mit der `shutdown`-Funktion die Ausführung des Programms beenden.

Eine beispielhafte Implementierung dieses Programms wird in Listing 4.3 veranschaulicht.

Vergleich

Die Vorgehensweisen bei Rust und X10 sind nahezu identisch. Der einzige nennenswerte Unterschied hierbei ist es, dass man in Rust keine explizite “Retreat“-Operation initiieren muss. Das C-Programm ist im Vergleich zu den beiden anderen Programmiersprachen hingegen komplexer und somit auch fehleranfälliger. Zudem benötigt ein solches C-Programm ungefähr doppelt so viele Codezeilen wie die äquivalenten Rust- oder X10-Programme. Obendrein ist es in C nicht möglich, Closures oder ähnliche Konstrukte zu verwenden, Rust und X10 unterstützen dies jedoch.

Listing 4.3: Minimales Invade, Infect, Retreat in C

```
#include <octopos.h>
#include <stdio.h>

void signaler(void* sig) {
    simple_signal* s = (simple_signal*)(sig);
    simple_signal_signal_and_exit(s);
}

void ilet_fn(void *signal) {
    printf("Hello World!\n");
    simple_ilet answer;
    simple_ilet_init(&answer, signaler, signal);
    dispatch_claim_send_reply(&answer);
}

void main_ilet(claim_t claim_id) {

    constraints_t constr = agent_constr_create();
    agent_constr_set_quantity(constr, 4, 4, 0);
    agent_constr_set_tile_shareable(constr, 1);
    agentclaim_t claim = agent_claim_invade(0, constr);

    simple_signal sync;
    simple_signal_init(&sync, agent_claim_get_pecount(claim));

    for (int tile=0; tile < get_tile_count(); tile++) {
        int pes=agent_claim_get_pecount_tile_type(claim,
tile, 0);
        if (pes) {
            proxy_claim_t p_claim =
                agent_claim_get_proxyclaim_tile_type(
                    claim, tile, 0
                );

            simple_ilet ilets[pes];
            for (int i = 0; i < pes; ++i) {
                simple_ilet_init(&ilets[i], ilet_fn, &sync);
            }

            proxy_infect(p_claim, &ilets[0], pes);
        }
    }

    simple_signal_wait(&sync);
    agent_claim_retreat(claim);
    shutdown(0);
}
```

5 Fazit und Ausblick

Im Folgenden werden die Ergebnisse der Evaluation bewertet und ein Ausblick auf die Zukunft der Programmiersprache Rust im Bezug zum invasiven Rechnen geboten.

5.1 Fazit

Rust eliminiert einige Fehlerquellen, welche in C oder anderen herkömmlichen Systemsprachen zu undefiniertem Verhalten führen können. In Rust muss der Programmierer nicht selbst auf solche Fehler achten und wird bereits vom Compiler auf Fehler hingewiesen. Dies erlaubt es, sicherere und weniger fehleranfällige Programme zu schreiben. Sollte es trotzdem während der Laufzeit zu schwerwiegenden Fehlern kommen, beispielsweise bei einem Pufferüberlauf, brechen Rust Programme die Ausführung ab und weisen so kein undefiniertes Verhalten auf.

Des Weiteren weist Rust in Situationen, in denen viele Objekte erstellt werden und den Geltungsbereich wieder verlassen, ein mindestens 10-fach besseres Laufzeitverhalten auf als X10. Zudem eignet sich X10 wegen des *Garbage Collectors* nicht für Anwendungsgebiete, in denen Pausen jeglicher Art nicht erwünscht sind, beispielsweise bei Echtzeitsystemen. In diesem Einsatzgebiet ist die Verfügbarkeit des Systems wichtig, daher wäre eine *Garbage Collector*-Pause nicht wünschenswert. Rust stellt hier eine empfehlenswertere Option dar.

Außerdem weist das Kompilieren mit *octorust* eine 100- bis 200-fach kürzere Kompilierungsdauer als der *x10i*-Compiler auf. Dies kann bei der Entwicklung von Programmen ein Zeitersparnis von mehreren Minuten pro Kompilierungsvorgang zur Folge haben. Zudem sind die Dateigrößen der kompilierten Rust Programme im betrachteten Minimalfall 6511716 Byte kleiner als die Größe eines kompilierten invasiven X10 Programms.

Rust bietet beim Programmieraufwand kaum Vorteile gegenüber X10, einzig das implizite *Retreat* wäre hier zu erwähnen. X10 bietet zudem derzeit robustere Abstraktionen über die C-Schnittstelle von *OctoPOS* und *iRTSS*. Vergleicht man Rust diesbezüglich allerdings mit C, so erkennt man eine Reduktion im Programmieraufwand.

Negativ zu betrachten wäre die Abhängigkeit von der *OctoPOS/iRTSS* C-Schnittstelle, welche die Sicherheit von Rust teils aufgibt. Die Funktionen der C-Schnittstelle bieten nämlich keine Garantien bezüglich der Sicherheit und die Nutzung von `void`-Zeigern zum Übertragen von Parametern ist ebenfalls ein Problem, welches zu Fehlern führen kann.

Außerdem weist Rust bei mathematischen Berechnungen keinen Vorteil gegenüber C oder X10 auf. In den betrachteten Messungen benötigte Rust meist mehr Zeit um dieselben Aufgaben zu lösen.

5.2 Ausblick

Da Rust erst im Jahre 2015 offiziel veröffentlicht wurde, sind noch nicht viele Bibliotheken für diese implementiert. Zwar können mithilfe des *Foreign Function Interface* Bibliotheken, die in anderen Sprachen geschrieben worden sind, eingebunden werden, diese bieten jedoch nicht die Sicherheitsgarantien wie Rust es tut. In dieser Hinsicht sollte sich die Lage jedoch in der Zukunft verbessern, wenn Rust weiterhin gerne von Entwicklern genutzt wird und eventuell großflächiger zum Einsatz kommt.

Eine weitere Möglichkeit in der Zukunft ist die Portierung der Standardbibliothek auf die SPARC-V8-Architektur. Die Standardbibliothek bietet einige hilfreiche Konstrukte und Funktionen, die das Programmieren erleichtern und sicherer machen. Möglich wäre es, dass die Entwickler der Rust Programmiersprache selbst die SPARC-V8-Architektur in der Zukunft unterstützen und so die Standardbibliothek portiert wird. Alternativ wäre es möglich, dass die Standardbibliothek im Rahmen des invasiven Rechnen portiert wird, sollte genügend Interesse daran bestehen.

Durch die unsichere Art und Weise, in der die Parameterübergabe in der *Infect*-Phase derzeit in *octolib* angeboten wird, können Programmfehler entstehen. Dies könnte beispielsweise durch den Einsatz von Serialisierungsbibliotheken wie **`serde`** in der Zukunft verbessert werden.

Um die negativen Effekte der Abhängigkeit von der *OctoPOS/iRTSS* C-Schnittstelle zu verringern, können weitere Abstraktionen erstellt werden, welche von den besonderen Eigenschaften der Rust-Programmiersprache Gebrauch machen. Dass dies möglich ist, hat die Implementierung des X10-Compilers *x10i* bewiesen, denn dieser verwendet dieselbe C-Schnittstelle, bietet jedoch zahlreiche Abstraktionen über diese und erlaubt so das Entwickeln von X10-Programmen, ohne dass der Programmierer sich dabei mit der C-Schnittstelle befassen muss.

Literaturverzeichnis

[1] R. R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, pp. 52–59, June 1997.

[2] L. B. Kish, "End of moore's law: thermal (noise) death of integration in micro and nano electronics," *Physics Letters A*, vol. 305, no. 3, pp. 144 – 149, 2002.

[3] NVIDIA, "Geforce gtx 1080 specifications (https://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-1080/specifications)," Zugriff: 2017-10-13.

[4] N. D. Matsakis and F. S. Klock, II, "The rust language," *Ada Lett.*, vol. 34, pp. 103–104, Oct. 2014.

[5] stackoverflow.com, "Developer survey results 2016 (https://insights.stackoverflow.com/survey/2016)," Zugriff: 2017-10-01.

[6] stackoverflow.com, "Developer survey results 2017 (https://insights.stackoverflow.com/survey/2017)," Zugriff: 2017-11-02.

[7] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, (New York, NY, USA), pp. 81–89, ACM, 2016.

[8] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?," in *Proceedings of the Asia-Pacific Workshop on Systems*, p. 9, ACM, 2012.

[9] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2017.

[10] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto, "Ownership is theft: experiences building an embedded os in rust," in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, pp. 21–26, ACM, 2015.

[11] J. A. Tov and R. Pucella, "Practical affine types," in *ACM SIGPLAN Notices*, vol. 46, pp. 447–458, ACM, 2011.

[12] rust lang.org, "Trait core::marker::copy (https://doc.rust-lang.org/1.8.0/core/marker/trait.copy.html)," Zugriff: 2017-10-29.

[13] S. Klabnik, "The rust programming language - first edition (https://doc.rust-lang.org/stable/book/first-edition/)," Zugriff: 2017-10-30.

[14] S. Klabnik, "The rust programming language - version 1.2.0 (https://doc.rust-lang.org/1.2.0/book/)," Zugriff: 2017-10-30.

[15] rust-lang nursery, "rustup.rs - the rust toolchain installer (https://github.com/rust-lang-nursery/rustup.rs)," Zugriff: 2017-10-29.

[16] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis, "Gpu programming in rust: Implementing high-level abstractions in a systems-level language," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pp. 315–324, May 2013.

[17] R. B. Garner, *The Scalable Processor Architecture (SPARC)*, pp. 3–31. New York, NY: Springer New York, 1991.

[18] J. Grobschadl, S. Tillich, and A. Szekely, "Performance evaluation of instruction set extensions for long integer modular arithmetic on a sparc v8 processor," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pp. 680–689, Aug 2007.

[19] J. Gaisler, "A portable and fault-tolerant microprocessor based on the sparc v8 architecture," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 409–415, 2002.

[20] V. Lari, "Invasive tightly coupled processor arrays," in *Invasive Tightly Coupled Processor Arrays*, pp. 21–81, Springer, 2016.

[21] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat, "Octopos: A parallel operating system for invasive computing," in *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, pp. 9–14, 2011.

[22] J. Teich, W. Schröder-Preikschat, and A. Herkersdorf, "Invasive computing-common terms and granularity of invasion," *arXiv preprint arXiv:1304.6067*, 2013.

[23] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe, "Invasive manycore architectures," in *17th Asia and South Pacific Design Automation Conference*, pp. 193–200, Jan 2012.

[24] R. K. Pujari, T. Wild, A. Herkersdorf, B. Vogel, and J. Henkel, "Hardware assisted thread assignment for risc based mpsocs in invasive computing," in *2011 International Symposium on Integrated Circuits*, pp. 106–109, Dec 2011.

[25] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau, "An x10 compiler for invasive architectures," Tech. Rep. 9, Karlsruhe Institute of Technology, 2012.

[26] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip – Hardware Design and Tool Integration* (M. Hübner and J. Becker, eds.), pp. 241–268, Springer, Berlin, Heidelberg, 2011.

[27] J. Aparicio, "initial sparc support (https://github.com/rust-lang/rust/pull/38314)," Zugriff: 2017-10-15.

[28] P. D. N. Henze and P.-D. D. D. Kadelka, *WAHRSCHEINLICHKEITSTHEORIE UND STATISTIK FÜR STUDIERENDE DER INFORMATIK und DES INGENIEURWESENS*. 2010.

Erklärung

Hiermit erkläre ich, Hermann Heinz Erich 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

Danksagung

Ich danke meinen Eltern, Janine und Heinz, als auch meinem Bruder Michael, die mich meine gesamtes Leben lang durch dick und dünn begleitet und unterstützt haben. 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 mein Studium am KIT nicht halb so schön wäre. Ich danke meiner „Ersatzfamilie“, der Familie Eherler, die immer einen Platz in ihrer Mitte für mich hat. Ich danke meinem Betreuer Andreas Zwinkau, welcher mich freundlich und hilfreich durch die Erstellung dieser Arbeit begleitet hat. Und zu guter Letzt danke ich dem Karlsruher Institut für Technologie, welches es mir erst ermöglichte, dieses Studium zu absolvieren.