Verständnis und Weiterentwicklung der Programmiersprache Rust[1]

Ralf Jung[2]

Abstract: *Rust* ist eine junge systemnahe Programmiersprache. Sie vereint die Sicherheit und das Abstraktionsniveau von Sprachen wie Java und Haskell mit der Kontrolle von Systemressourcen, wie C und C++ sie bieten. Meine Dissertation [Ju20a] untersucht die Sicherheitsgarantien von Rust erstmals formell und trägt somit entscheidend zum besseren *Verständnis* und zur *Entwicklung* dieser zunehmend bedeutsamen Sprache bei. Dafür habe ich drei Systeme entwickelt und im Beweisassistenten Coq verifiziert: RustBelt, Iris, und Stacked Borrows.

RustBelt ist ein formelles Modell des Typsystems von Rust einschließlich eines Korrektheitsbeweises, welcher die Sicherheit von Speicherzugriffen und Nebenläufigkeit zeigt. RustBelt ist in der Lage, einige komplexe Komponenten der Standardbibliothek von Rust zu verifizieren, obwohl die Implementierung dieser Komponenten intern *unsichere* Sprachkonstrukte verwendet.

RustBelt ist nur möglich dank der Entwicklung von *Iris*, einem Framework zur Konstruktion von Separationslogiken zur Programmverifikation von beliebigen Programmiersprachen. Die Stärke von Iris liegt in der Möglichkeit, neue Beweismethoden mit Hilfe weniger einfacher Bausteine herzuleiten.

Stacked Borrows ist eine Erweiterung der Spezifikation von Rust, die es dem Compiler erlaubt, den Quelltext mit Hilfe der im Typsystem kodierten Alias-Informationen besser zu analysieren. So werden neue mächtige intraprozedurale Optimierungen ermöglicht.

1 Einführung

Im Bereich der Systemprogrammierung genießen Sprachen ohne starke Typ- und Speichersicherheit nach wie vor eine große Verbreitung. Ein Großteil der Software, die das Fundament moderner Computer bildet, ist in C oder C++ geschrieben – Sprachen, die sich über die Jahre deutlich weiterentwickelt haben, aber nach wie vor die Verantwortung für grundlegende Speichersicherheit dem Programmierer überlassen. Programmierer jedoch machen unvermeidlich Fehler, und das mit handfesten Konsequenzen: sowohl Microsoft als auch die Entwickler von Google Chrome geben an, dass ca. 70% der Sicherheitslücken in ihren Produkten durch Verletzungen der Speichersicherheit entstehen [Th19, Ch20].

Viele Sprachen erzielen Speichersicherheit, indem sie zu einem gewissen Grad dem Programmierer die Kontrolle darüber entziehen, wie das Programm mit dem Speicher interagiert. Das gilt insbesondere für das automatische Bereinigen des Speichers durch einen "garbage collector", wobei sowohl die Struktur der Daten im Speicher als auch die Deallokation von nicht mehr benötigtem Speicher aus der Hand des Programmierers genommen werden. Bei der Programmierung von fundamentalen Systemkomponenten ist dies jedoch

[1] Englischer Titel der Dissertation: "Understanding and Evolving the Rust Programming Language"

[2] MPI-SWS, jung@mpi-sws.org

keine Option. Hier sind minimaler Speicher- und Rechenzeitverbrauch oberste Priorität, und der Programmierer muss diese Aspekte direkt unter Kontrolle haben. Daher wird eine Sprache benötigt, die in dieser Hinsicht mit C und C++ auf einer Ebene steht, während sie gleichzeitig Speichersicherheit und Typsicherheit garantiert.

Rust beansprucht für sich, solch eine Sprache zu sein. Seinen Ursprüngen bei Mozilla entwachsen, lebt Rust inzwischen von einer großen Open-Source-Gemeinschaft und wird zunehmend auch industriell eingesetzt: in Mozillas Firefox, aber auch in vielen anderen Firmen vom kleinen Start-Up bis zu Tech-Riesen wie Amazon und Microsoft.[3]

Ähnlich wie C++ hat der Programmierer bei Rust volle Kontrolle über die Struktur und Deallokation von Daten im Speicher. Eine weitere Parallele ist der Fokus auf "zero-cost" Abstraktionen im Sinne von Stroustrup [St94]: "What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better."

Im großen Gegensatz zu C++ jedoch verspricht Rust Typsicherheit und Speichersicherheit. Rust will außerdem Probleme aus der Welt schaffen, unter denen auch viele Sprachen mit Speichersicherheit leiden, zum Beispiel Iteratoren, die ihre Gültigkeit durch gleichzeitige Veränderung der zugrundeliegenden Datenstruktur verlieren. Darüber hinaus nimmt Rust für sich in Anspruch, Nebenläufigkeitsfehler zu vermeiden: Rust-Programme haben keine "data races", d.h. es gibt keine *unbeabsichtigte* Kommunikation zwischen mehreren Threads durch das Verwenden desselben Speicherbereichs. Damit geht Rust über die Sicherheitsgarantien der meisten "sicheren" Sprachen hinaus.[4]

Soweit die Behauptungen, mit denen Rust von sich Reden macht. Doch ist Rust wirklich in der Lage, diesen Ansprüchen gerecht zu werden? Im Rahmen meiner Dissertation [Ju20a] habe ich das erste logische Framework entwickelt, das in der Lage ist, diese Ansprüche formell zu beweisen. Im Folgenden werde ich den grundlegenden Ansatz dieses Beweises erläutern; zuerst jedoch will ich darlegen, was diesen Beweis so anspruchsvoll macht.

2 Mehr Sicherheit trotz "unsafe" Code?

Das folgende Beispiel zeigt repräsentativ, welche Art von Problemen in C++-Programmen auftreten können:

```
1  std::vector<int> v { 10, 11 };
2  int *vptr = &v[1]; // Zeigt auf den *Inhalt* von ‘v‘.
3  v.push_back(12); // Verschiebt den Inhalt von ‘v‘ an eine andere Stelle.
4  std::cout << *vptr; // Alter Inhalt von ‘v‘ nach Deallokation verwendet.
```

In der ersten Zeile wird ein `std::vector<int>`, also ein vergrößerbares Array von Ganzzahlen, angelegt. Der Inhalt von `v`, die beiden Elemente `10` und `11`, werden in einem dafür

[3] Unter `https://www.rust-lang.org/production/users` sind Firmen aufgelistet, die Rust in ihren Produkten einsetzen; unter `https://foundation.rust-lang.org/members` sind die Mitglieder der kürzlich gegründeten Rust Foundation zu sehen.

[4] In Java zum Beispiel sind data races möglich, und es gibt nur schwache Garantien, was in diesem Fall passiert. In Go und Swift können data races sogar die Speichersicherheit verletzen.

allozierten Bereich im Speicher abgelegt. In der zweiten Zeile wird ein Zeiger `vptr` erstellt, der *in* diesen Bereich zeigt; genau genommen zeigt er auf das zweite Element (welches aktuell den Wert `11` hat). `v` und `vptr` zeigen jetzt beide auf überlappende Teile desselben Speicherbereichs, es besteht also *Aliasing* zwischen diesen Zeigern.

In Zeile 3 wird `v` um ein neues Element am Ende verlängert: `12` wird nach der `11` in den Speicherbereich mit dem Inhalt von `v` abgelegt. Falls dafür nicht mehr genug Platz ist, wird ein neuer Speicherbereich alloziert, die vorhandenen Elemente werden dorthin verschoben und der alte Bereich wird dealloziert. Für dieses Beispiel gehen wir davon aus, dass genau das passiert. Dieser Fall ist besonders interessant, weil `vptr` noch auf den alten Speicherbereich zeigt! Mit anderen Worten, durch das Hinzufügen eines neuen Elements zu `v` wurde der Zeiger `vptr` ungültig. Das wird in der letzten Zeile zum Problem. Hier greift das Programm mit Hilfe des Zeigers auf den alten, deallozierten Speicher zu: ein klassischer "use-after-free"-Fehler.

Das Beispiel mag künstlich wirken, aber in der Praxis wird der Aufruf von `push_back` an einer völlig anderen Stelle im Code sein als `vptr`. Statt eines expliziten Zeigers ist `vptr` oft ein Iterator; man spricht dann auch von "iterator invalidation". Insbesondere bei der Wartung von vorhandenem Code ist es oft quasi unmöglich festzustellen, ob ein `push_back` irgendwelche wichtigen Zeiger an anderer Stelle im Programm ungültig macht.

In Rust werden solche Probleme statisch erkannt – an Stelle eines Laufzeitfehlers oder einer Sicherheitslücke gibt es eine Fehlermeldung vom Compiler. Die Rust-Version des C++-Programms sieht wie folgt aus;

```
1  let mut v: Vec<i32> = vec![10, 11];
2  let vptr = &mut v[1]; // Zeigt auf den *Inhalt* von ‘v‘.
3  v.push(12); // Verschiebt den Inhalt von ‘v‘ an eine andere Stelle.
4  println!("{}", *vptr); // Fehlermeldung durch Compiler.
```

Wie gehabt gibt es hier einen getrennten Bereich im Speicher, in dem die Elemente von `v` gespeichert werden; und wie gehabt kann `push` diesen Speicherbereich verschieben, was `vptr` ungültig macht und in der letzten Zeile zu einem Problem würde – wenn nicht der Compiler das Programm mit dem Fehler ablehnte, dass `v` nicht "mehrfach zur selben Zeit veränderlich ausgeliehen" werden kann. In meiner Dissertation erkläre ich im Detail, wie der Rust-Compiler dieses Problem erkennt. Für diese Einführung genügt es zu wissen, dass das Typsystem von Rust komplizierter ist als bei anderen Sprachen üblich und Ideen wie *Eigentum* (eine Form von linearen Typen) involviert sowie eine Datenflussanalyse, die *Leihgaben* ("borrows") von Zeigern für gewisse *Lebenszeiten* ("lifetimes") ermöglicht.

Allerdings hat dieser Ansatz eine erhebliche Einschränkung: Datenstrukturen wie `Vec` verwenden Code, den auch das komplizierte Typsystem von Rust nicht vollständig prüfen kann. Statt dessen gibt es in Rust das Schlüsselwort `unsafe`, welches syntaktisch Bereiche des Programms markiert, in denen gefährliche Operationen durchgeführt werden können. So wird sichergestellt, dass Programmierer nicht versehentlich den sicheren Bereich von Rust verlassen. `Vec` jedoch nutzt `unsafe`, um mit Hilfe ungeschützter Zeiger das Hinzufügen von Elementen in amortisiert konstanter Zeit zu realisieren (genau wie `std::vector` in C++).

Damit stellt sich natürlich die Frage: macht dieser `unsafe` Code nicht die Sicherheitsgarantien von Rust zunichte? Um das zu verhindern, bedient Rust sich des Konzepts der *Kapselung*. Das Typsystem kann sicherstellen, dass auch `unsafe` Code "nach außen", also bei Benutzung seiner öffentlichen Schnittstelle, sicher zu verwenden ist. Für die Sicherheit der Implementierung ist jedoch der Programmierer selbst verantwortlich.

Dieser Ansatz kann gut am Beispiel von `Vec` erläutert werden. In C++ gibt es für Typen wie `std::vector` ausführliche Dokumentation, welche erklärt, wie man den Typ korrekt benutzt und welche auf Probleme wie die in dem obigen Beispiel hinweist. Allerdings sind dies alles nur *Kommentare* – für den Compiler gibt es keine Möglichkeit, den Programmierer beim Einhalten dieser Regeln zu unterstützen. Im Gegensatz dazu sind bei `Vec` in Rust die *Typen* der beteiligten Funktionen detailliert genug, dass der Compiler prüfen kann, ob der Nutzer sich an die angegebenen Regeln hält. Die Autoren von `Vec` versprechen, dass die internen `unsafe` Operationen von diesen Typen "gekapselt" werden: Programmierer, die nur die öffentliche Schnittstelle zu `Vec` verwenden und selber keinen Gebrauch von `unsafe` machen, genießen weiterhin die volle Typ- und Speichersicherheit von Rust. Das Typsystem von Rust ist nicht stark genug, um korrekte "Kapselung" zu prüfen, aber es *ist* stark genug, um zu prüfen, ob die öffentliche Schnittstelle *korrekt verwendet* wird.

Dabei bleibt jedoch ein Problem: Die öffentliche Schnittstelle von `Vec` basiert auf den Konzepten von Eigentum und Leihgaben, die dem Rust-Typsystem zugrunde liegen. Intuitiv sind diese Konzepte gut verstanden, aber der Teufel steckt wie üblich im Detail: Sind die Invarianten des Typsystem tatsächlich stark genug, um die korrekte Verwendung von `Vec` zu garantieren? Bei `Vec` ist das relativ unstrittig; die Interaktion mit dem Typsystem ist hier nicht kompliziert. Andere Komponenten der Standardbibliothek, wie zum Beispiel `Mutex`, verwenden das Typsystem jedoch auf deutlich interessantere Weise. `Mutex` erlaubt es Rust-Code, veränderliche Daten zwischen mehreren Threads zu teilen, wobei durch ein Lock sichergestellt wird, dass immer nur ein Thread gleichzeitig auf den Daten arbeitet. Im Allgemeinen geht der Rust-Compiler davon aus, dass geteilte Daten nicht verändert werden können, aber Typen wie `Mutex` verwenden eine subtile Kombination von getypter Schnittstelle und Laufzeitkontrollen, um diese Einschränkung zu umgehen und so Rust-Code das sichere Arbeiten mit geteilten veränderlichen Daten zu ermöglichen. Es ist alles andere als offensichtlich, dass `Mutex` auf diese Art nicht die Sicherheitsgarantien von Rust verletzt. Um Rusts Versprechen von sicherer Systemprogrammierung einzulösen, ist es also wichtig, dies formell prüfen zu können.

Allerdings ist der übliche *syntaktische* Ansatz zum Beweis von Typsicherheit [WF94] für Rust nicht geeignet. Bei diesem Ansatz geht man von einer geschlossenen Welt aus, man nimmt also an, dass dem Programm nur einen fester Satz von Primitiven mit ihren Typregeln zur Verfügung steht. Typsicherheit in Rust kann so nur für Programme gezeigt werden, die keinerlei `unsafe` Code verwenden, auch nicht indirekt über eingebundene Bibliotheken. Natürlich gilt diese Einschränkung auch für andere Programmiersprachen, deren Typsystem man mit einem Konstrukt wie `unsafe` umgehen kann, wie zum Beispiel OCaml mit `Obj.magic`, Haskell mit `unsafePerformIO`, oder jede beliebige Sprache von der aus man C-Bibliotheken aufrufen kann. Allerdings werden solche Konstrukte in anderen Sprachen bei formeller Betrachtung der Typsicherheit üblicherweise ignoriert. Diese Lücke in

den Beweisen ist nicht zufriedenstellend, und sie wäre in Rust noch deutlich größer als bei den meisten anderen Sprachen: Um eine mit C oder C++ vergleichbare Performance zu erreichen, sind bei grundlegenden Datenstrukturen keine Kompromisse möglich. `unsafe` wird daher in den unteren Schichten des Rust-Ökosystems sehr viel eingesetzt, und jede realistische Betrachtung von Rust muss sich auch mit `unsafe` auseinandersetzen.

3 RustBelt: Ein tieferes Verständnis von Rust

In meiner Dissertation beschreibe ich RustBelt [Ju18a], das erste formelle (und maschinengeprüfte) Modell von Rust, welches in der Lage ist, Typsicherheit bei Verwendung von `unsafe` zu beweisen. Dieser Beweis ist *erweiterbar* für eine neue Bibliothek mit interner Verwendung von `unsafe`, und zwar in dem Sinne, dass RustBelt klar definiert, welche Aussage bewiesen werden muss, damit alle Nutzer dieser Bibliothek weiterhin Speichersicherheit und Threadsicherheit genießen. Der Beweis ist außerdem *modular* in dem Sinne, dass mehrere Bibliotheken unabhängig voneinander verifiziert werden können und der Beweis auch bei beliebiger Kombination ihrer Schnittstellen seine Gültigkeit behält.

Die zentrale Idee hinter diesem Beweis ist die, ein *semantisches Modell* von Rusts Typsystem zu definieren. Dies ist ein durchaus bewährter Ansatz, der bereits für den allerersten Typsicherheitsbeweis von Milner [Mi78] für einen polymorphen λ-Kalkül im Stile von ML eingesetzt wurde. Milners Ansatz basiert auf früheren Arbeiten an *logischen Relationen* [Ta67]. Logische Relationen definieren die *Bedeutung* eines Typs als die Menge der Werte, die das gewünschte *beobachtbare Verhalten* an den Tag legen. Insbesondere ist für den Typ einer Funktion nur ihr Eingabe-Ausgabe-Verhalten relevant, im Gegensatz zum syntaktischen Ansatz, wo der Quelltext der Funktion gewissen Regeln genügen muss. Beim semantischen Ansatz darf die Funktion durchaus potentiell unsichere Dinge tun, solange die Garantien und Invarianten des Typsystems dabei nicht verletzt werden.

Es gelang zunächst nicht, diesen Ansatz auf mächtigere Sprachen mit Seiteneffekten und Funktionen höherer Ordnung anzuwenden, weshalb sich der einfachere (aber weniger mächtige) syntaktische Ansatz von “progress and preservation” durchsetzte. Allerdings gab es in den letzten zwei Jahrzehnten große Fortschritte auf dem Gebiet der logischen Relationen [Ah04], sodass es inzwischen prinzipiell möglich ist, den semantischen Ansatz mit den ausdrucksstarken Typsystemen moderner Sprachen zu verwenden. In meiner Arbeit wende ich diesen Ansatz nun erstmals auf ein Typsystem wie das von Rust an.

Nach der Definition des semantischen Modells sind drei Schritte nötig, um den Beweis der Typsicherheit in RustBelt zu vervollständigen:

1. Das *fundamentale Theorem der logischen Relation* stellt sicher, dass alle syntaktischen Typregeln korrekt sind, wenn man sie semantisch interpretiert. Dafür wird nicht nur den Typen, sondern auch allen anderen Komponenten der Typregeln eine semantische “Bedeutung” zugeordnet, welche (intuitiv gesprochen) die durch das Typsystem kodierten Invarianten explizit macht.
2. Zudem muss man beweisen, dass ein *semantisch* wohlgetyptes Programm in der Tat speichersicher und threadsicher ist.

3. Schließlich müssen die Bibliotheken, die intern `unsafe` verwenden, korrekt bewiesen werden. Das semantische Modell ermöglicht es, die Typen der öffentlichen Schnittstelle dieser Bibliotheken in einen formellen Vertrag umzuwandeln. Die Herausforderung besteht nun darin, zu beweisen, dass die Implementierung der Schnittstelle dem Vertrag genügt.

Zusammengenommen zeigen diese Schritte, dass ein Programm speicher- und threadsicher ist, wenn aller `unsafe` Code sich innerhalb von korrekt bewiesenen Bibliotheken befindet.

Für den dritten Schritt habe ich viele komplexe Komponenten der Standardbibliothek korrekt bewiesen, die eine zentrale Rolle im Rust-Ökosystem spielen, insbesondere solche zum Arbeiten mit geteilten veränderlichen Daten: `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex` (hier wurde durch meine Arbeit ein Fehler aufgedeckt und behoben), `RwLock`, `mem::swap` und `thread::spawn`; sowie `rayon::join` und `take_mut`, welche weitere interessante Aspekte des Typsystems aufzeigen. Alle diese Beweise habe ich mit dem Beweisassistenten Coq durchgeführt; sie wurden also von Coq auf Korrektheit geprüft.

Die Entwicklung eines semantischen Modells für ein Typsystem wie das von Rust stellte mich vor einige größere technische Herausforderungen. In dieser Zusammenfassung möchte ich kurz auf die beiden größten Hürden eingehen: die *Wahl der richtigen Logik* und die Entwicklung eines Modells für *Leihgaben und Lebenszeiten*.

Die Wahl der Logik. "Welche Logik wird verwendet" mag nach einer seltsamen Frage klingen, aber für das semantische Modell von RustBelt war dies in der Tat eine der wichtigsten Entscheidungen. In Rust drücken Typen nicht nur aus, dass ein Wert eine bestimmte Form hat, sondern decken auch Aspekte des *Eigentums* an den verwendeten Ressourcen ab, zum Beispiel die exklusive Kontrolle über einen bestimmten Speicherbereich. Eigentum kann explizit beim Bau des semantischen Modells in Betracht gezogen werden, aber dieser Ansatz ist mühselig und fehleranfällig, vergleichbar mit dem Schreiben eines Programms in Assemblersprache. RustBelt verwendet Separationslogik ("separation logic") [Re02], um auf einem höheren Abstraktionsniveau arbeiten zu können. Separationslogik kann direkt logische Aussagen über Eigentum von Speicherbereichen treffen und eignet sich daher gut dafür, das Eigentum von Typen wie `Vec` zu definieren.

Allerdings ist Eigentum von Speicherbereichen allein nicht ausreichend. Um die Korrektheit von Rust-Typen wie `Mutex` zu beweisen, sind flexiblere Formen von Eigentum notwendig. Für solche Zwecke haben wir *Iris* [Ju15, Ju16, Kr17, Ju18b] entwickelt, eine hochgradig flexible Separationslogik wo Nutzer ihre eigenen Formen von "Eigentum" definieren können. Iris unterstützt außerdem "step-indexing" [DAB11], eine wichtige Komponente moderner logischer Relationen, und nimmt dem Nutzer weitgehend die üblicherweise damit verbundene Buchführung ab. Zu guter Letzt ermöglicht Iris interaktive maschinengeprüfte Beweise in Coq [Kr18].

Leihgaben und Lebenszeiten. Für ein vollständiges Modell des Typsystems von Rust wird eine Logik benötigt, die in der Lage ist, nicht nur über Eigentum zu argumentieren, sondern auch über *Leihgaben* von Eigentum für eine gewisse *Lebenszeit*. Hier stellte es

sich als entscheidend heraus, dass Iris von Anfang an darauf ausgelegt war, dem Nutzer das Herleiten neuer Beweismethoden in der Logik möglichst einfach zu machen.

Unter Verwendung aller wichtigen Komponenten von Iris, insbesondere *impredikativer Invarianten* [SB14] und *Ressourcen höherer Ordnung* [Ju16], habe ich eine solche "Lebenszeitlogik" in Iris definiert und ihre Korrektheit beweisen. Die Lebenszeitlogik ermöglicht es, bei der Definition des semantischen Modells von Rust-Typen direkten Gebrauch vom Konzept einer Leihgabe zu machen, und auch der Korrektheitsbeweis des Typsystems kann auf diesem hohen Abstraktionsniveau geführt werden.

4 Stacked Borrows: Mehr Optimierungen für Rust

Typsysteme wie das von Rust sind nicht nur nützlich, weil sie Programme sicherer und zuverlässiger machen, sie können auch dabei helfen, effizienteren Code zu erzeugen. Zum Beispiel muss eine Sprache mit einem starken Typsystem nicht Rechenzeit und Speicher darauf aufwenden, dynamische Typinformationen zu verwalten. In Rust erzwingt das Typsystem eine strikte Aliasing-Disziplin auf Zeigern. Es wäre daher sehr interessant, diese statisch bekannte Information für Optimierungen auszunutzen.

So sind zum Beispiel *veränderliche Referenzen*, geschrieben `&mut T`, in Rust immer exklusiv in dem Sinne, dass aktuell keine anderen Zeiger auf dieselben Daten verwendet werden können. Das sollte es uns erlauben, die folgende Funktion zu optimieren:

```
1  fn example1(x: &mut i32, y: &mut i32) -> i32 {
2    *x = 42;
3    *y = 13;
4    return *x; // Hier wird 42 gelesen, weil x und y nicht aliasen!
5  }
```

`x` und `y` sind als veränderliche Referenzen beide exklusiv und können daher nicht aliasen, d.h. die Speicherbereiche, auf die sie zeigen, können nicht überlappen. Daher sollte dem Compiler die Annahme gestattet sein, dass in Zeile 4 immer `42` gelesen wird; der Speicherzugriff kann also durch eine Konstante ersetzt werden.

In Rust wird die Situation jedoch durch `unsafe` Code verkompliziert, welcher mittels direkter Zeigermanipulation die üblichen Alias-Regeln umgehen kann. Es ist nicht schwer, `unsafe` Code zu schreiben, welcher dazu führt, dass die obige Funktion `13` zurückgibt:[5]

```
1  fn main() {
2    let mut local = 5;
3    let raw_pointer = &mut local as *mut i32;
4    let result = unsafe {
5      example1(&mut *raw_pointer, &mut *raw_pointer)
6    };
7    println!("{}", result); // Ausgabe: "13".
8  }
```

[5] Diese Tests wurden mit Rust 1.35.0 im "release mode" durchgeführt.

In Zeile 3 wird die *Referenz* vom Typ `&mut i32` in einen *ungeschützten Zeiger* ("raw pointer") vom Typ `*mut i32` konvertiert. Wie bei Zeigern in C kann man ungeschützte Zeiger in Rust in Ganzzahlen konvertieren und umgekehrt, und auch Zeigerarithmetik ist möglich. Um Speichersicherheit nicht zu gefährden, ist das *Dereferenzieren* solcher Zeiger nur innerhalb von `unsafe`-Blöcken erlaubt; der Programmierer muss also explizit angeben, potentiell gefährliche Operationen zu verwenden, und haftet an dieser Stelle selber für die Typsicherheit. Ungeschützte Zeiger sind bei der Interaktion mit C-Bibliotheken notwendig und für eine effiziente Implementierung von Datenstrukturen wie `Vec`.

Dieses Beispiel jedoch nutzt ungeschützte Zeiger, um das Typsystem gezielt zu untergraben. In Zeile 5 wird der Zeiger zurück in eine Referenz umgewandelt, indem man ihn dereferenziert und dann direkt eine neue Referenz erstellt (`&mut *raw_pointer`). Und weil das Typsystem diese Zeiger kaum kontrolliert, kann dies auch zweimal geschehen! Im Endeffekt wird also `example1` mit zwei Referenzen auf dieselbe Variable aufgerufen: es besteht Aliasing zwischen `x` und `y`, was eigentlich unmöglich sein sollte. `example1` gibt dementsprechend `13` zurück, und wenn das Programm so optimiert würde, dass es immer `42` zurückgibt, würde sich das beobachtbare Verhalten des Programms verändern. Damit ist die Optimierung in diesem Fall also inkorrekt.

Es ist an dieser Stelle verlockend, das Problem zu ignorieren, da es ja "nur" `unsafe` Code betrifft. Dies wird jedoch der (oben bereits angesprochenen) wichtigen Rolle von `unsafe` Code im Rust-Ökosystem nicht gerecht. Damit die Optimierung auch bei Verwendung von `unsafe` durchgeführt werden kann, muss vom Programmierer verlangt werden, dass `unsafe` Code das Typsystem nicht wie oben geschehen untergräbt. Doch was genau sind die Bedingungen, die `unsafe` Code dafür erfüllen muss?

Als Antwort auf diese Frage beschreibe ich im dritten Teil meiner Dissertation *Stacked Borrows* [Ju20b], eine operationale Semantik für das Aliasing von Zeigern in Rust. Gemäß dieser Semantik hat das Beispielprogramm *undefiniertes Verhalten*, es gilt also als ungültig.[6] Der Compiler muss für ungültige Programme keine korrekte Ausführung sicherstellen, sodass hier also kein Gegenbeispiel mehr vorliegt. Gleichzeitig gilt: gültiger `unsafe` Code mit voll definiertem Verhalten wird durch die Optimierung nicht beeinflusst.

Im Vergleich zu einer naiven Semantik, wie sie in RustBelt verwendet wird, fügt Stacked Borrows eine neue Form von undefiniertem Verhalten hinzu: die Verletzung der Aliasing-Regeln. Bei undefiniertem Verhalten ist allerdings Speichersicherheit nicht mehr gewährleistet, daher darf es nicht "zu viel" undefiniertes Verhalten geben. Alle Programme im typsicheren Fragment von Rust (ohne `unsafe`) müssen also weiterhin voll definiert sein, und es muss auch weiterhin möglich sein, Datenstrukturen wie `Vec` mit Hilfe von ungeschützten Zeigern zu definieren. Stacked Borrows wurde daher auf zwei Arten validiert:

- Um sicherzustellen, dass nicht zu viel undefiniertes Verhalten eingeführt wurde, habe ich Miri,[7] einen bereits vorhandenen Interpreter für Rust-Programme, mit einer direkten Implementierung der operationalen Semantik von Stacked Borrows ausgestattet. Anschließend habe ich große Teile der Test-Suite der Rust-Standardbibliothek

[6] Dies ist vergleichbar mit einem C-Programm, das z.B. einen Null-Zeiger dereferenziert.
[7] Mehr Informationen zu Miri gibt es online unter `https://github.com/rust-lang/miri/`.

in diesem Interpreter ausgeführt. So konnte ich prüfen, ob all diese Tests den strikten Regeln von Stacked Borrows genügen. Die überwiegende Mehrzahl der Tests brauchte dazu keinerlei Anpassungen. Ich habe jedoch auch einige Verletzungen der Alias-Regeln gefunden, von denen fast alle durch die Rust-Entwickler als Fehler anerkannt und inzwischen behoben wurden.

- Um sicherzustellen, dass alle potentiellen Gegenbeispiele als ungültig erklärt wurden, zeige ich in meiner Dissertation einige Beweisskizzen, welche die Korrektheit von Optimierungen wie der in `example1` für alle gültigen Programme belegen. Diese Beweisskizzen sind von maschinegeprüften Beweisen in Coq untermauert.

5 Schlussfolgerungen und Anwendungen

In meiner Dissertation beschreibe ich drei Projekte, die signifikant zur Entwicklung von interaktiver Programmverifikation allgemein und von Rust im Besonderen beitragen.

Iris, ein Framework zur Entwicklung von flexiblen Separationslogiken, erfährt bereits erste industrielle Nutzung und diente als Grundlage für mehr als 30 weitere Veröffentlichungen von Forschern an zehn verschiedenen Einrichtungen.[8]

RustBelt ist der erste formelle Beweis der Typsicherheit von Rust und Teilen seiner Standardbibliothek. Damit belegt es die Praktikabilität von semantischen Modellen für realistische Sprachen mit komplexen Typsystemen und stellt gleichzeitig die Sicherheit von Rust auf ein solides Fundament.

Stacked Borrows ist der erste konkrete Vorschlag für ein System von Aliasing-Regeln in Rust und dient auch dank meiner engen Zusammenarbeit mit dem Rust-Team aktuell als de-facto Standard für die korrekte Verwendung ungeschützter Zeiger. Die von mir entwickelte Erweiterung für Miri wird vielfach eingesetzt, um die Konformität von `unsafe` Code mit diesen Regeln zu prüfen. Damit ist die Grundlage geschaffen, um Stacked Borrows letztendlich zu einem offiziellen Teil der Rust-Spezifikation weiterzuentwickeln.

Literaturverzeichnis

[Ah04] Ahmed, Amal: Semantics of Types for Mutable State. Dissertation, Princeton University, 2004.

[Ch20] Chromium project, The: Chromium Security: Memory safety. Blog post, https://www.chromium.org/Home/chromium-security/memory-safety, 2020.

[DAB11] Dreyer, Derek; Ahmed, Amal; Birkedal, Lars: Logical Step-Indexed Logical Relations. LMCS, 7(2:16):1–37, Juni 2011.

[Ju15] Jung, Ralf; Swasey, David; Sieczkowski, Filip; Svendsen, Kasper; Turon, Aaron; Birkedal, Lars; Dreyer, Derek: Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In: POPL. S. 637–650, 2015.

[Ju16] Jung, Ralf; Krebbers, Robbert; Birkedal, Lars; Dreyer, Derek: Higher-order ghost state. In: ICFP. S. 256–269, 2016.

[8] Siehe https://iris-project.org/ für weitere Informationen zu Iris.

[Ju18a] Jung, Ralf; Jourdan, Jacques-Henri; Krebbers, Robbert; Dreyer, Derek: RustBelt: Securing the Foundations of the Rust Programming Language. PACMPL, 2(POPL):66:1–66:34, 2018.

[Ju18b] Jung, Ralf; Krebbers, Robbert; Jourdan, Jacques-Henri; Bizjak, Aleš; Birkedal, Lars; Dreyer, Derek: Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. JFP, 28:1–73, November 2018.

[Ju20a] Jung, Ralf: Understanding and Evolving the Rust Programming Language. Dissertation, Universität des Saarlandes, 2020.

[Ju20b] Jung, Ralf; Dang, Hoang-Hai; Kang, Jeehoon; Dreyer, Derek: Stacked Borrows: An Aliasing Model for Rust. PACMPL, 4(POPL), 2020.

[Kr17] Krebbers, Robbert; Jung, Ralf; Bizjak, Aleš; Jourdan, Jacques-Henri; Dreyer, Derek; Birkedal, Lars: The Essence of Higher-Order Concurrent Separation Logic. In: ESOP. Jgg. 10201 in LNCS, S. 696–723, 2017.

[Kr18] Krebbers, Robbert; Jourdan, Jacques-Henri; Jung, Ralf; Tassarotti, Joseph; Kaiser, Jan-Oliver; Timany, Amin; Charguéraud, Arthur; Dreyer, Derek: MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. PACMPL, 2(ICFP):77:1–77:30, 2018.

[Mi78] Milner, Robin: A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348–375, 1978.

[Re02] Reynolds, John C.: Separation logic: A logic for shared mutable data structures. In: LICS. S. 55–74, 2002.

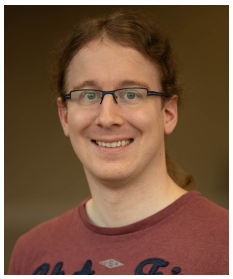
[SB14] Svendsen, Kasper; Birkedal, Lars: Impredicative Concurrent Abstract Predicates. In: ESOP. Jgg. 8410 in LNCS, S. 149–168, 2014.

[St94] Stroustrup, Bjarne: The Design and Evolution of C++. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[Ta67] Tait, W. W.: Intensional interpretations of functionals of finite type I. Journal of Symbolic Logic, 32(2), 1967.

[Th19] Thomas, Gavin: A proactive approach to more secure code. Blog post, https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/, 2019.

[WF94] Wright, Andrew K; Felleisen, Matthias: A syntactic approach to type soundness. Information and computation, 115(1), 1994.



Ralf Jung wurde am am 25. April 1990 in Wiesbaden, Deutschland, geboren. Er studierte Informatik an der Universität des Saarlandes und promovierte 2020 am MPI-SWS mit "summa cum laude". Seine Bachelorarbeit wurde mit dem FdSI-Bachelor-Preis für hervorragende Studienleistungen ausgezeichnet. In der Rust-Gemeinschaft ist er ein weithin anerkannter Experte für `unsafe` Code und leitet dort die Arbeitsgruppe zu "`unsafe` code guidelines". Er bloggt regelmäßig zu diesem Thema[9] und wirkt aktiv bei der Entwicklung der Sprache mit, unter anderem durch zwei Praktika bei Mozilla Research während seiner Promotionszeit. Derzeit arbeitet er als Post-Doc am MPI-SWS und ist zudem als externer Mitarbeiter eng in die PDOS-Forschungsgruppe am MIT eingebunden.

[9] Siehe https://www.ralfj.de/blog/categories/rust.html