Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

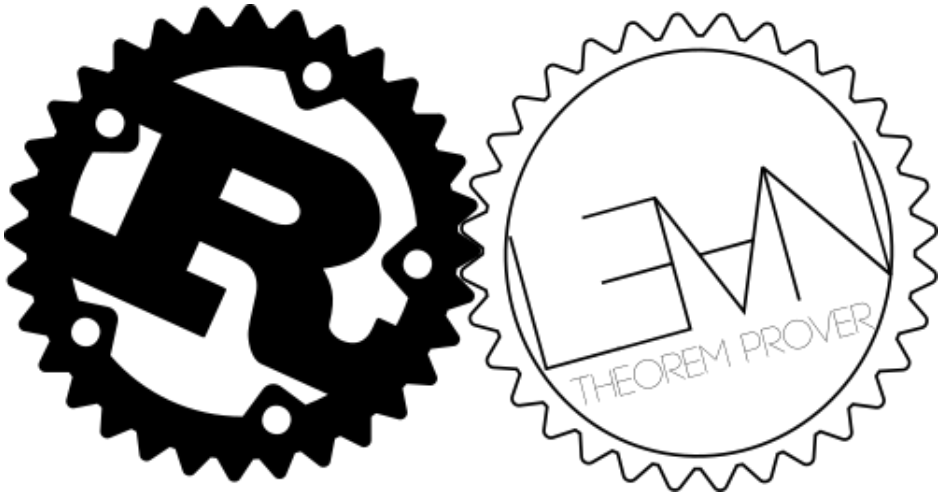
Lehrstuhl Prof. Dr.-Ing. Snelting

Simple Verification of Rust Programs via Functional Purification

Masterarbeit von

Sebastian Ullrich

an der Fakultät für Informatik



Erstgutachter: Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter: ???

Einfache Verifikation von Rust-Programmen

Imperative Programmiersprachen sind in der modernen Softwareentwicklung allgegenwärtig, stellen aber ein Hindernis für formale Softwareverifikation dar durch ihre Verwendung von veränderbaren Variablen und Objekten. Programme in diesen Sprachen können normalerweise nicht direkt auf die unveränderliche Welt von Logik und Mathematik zurückgeführt werden, sondern müssen in eine explizit modellierte Semantik der jeweiligen Sprache eingebettet werden. Diese Indirektion stellt ein Problem für die Benutzung von interaktiven Theorembeweisern dar, da sie die Entwicklung von neuen Werkzeugen, Taktiken und Logiken für diese "innere" Sprache bedingt.

Die vorliegende Arbeit stellt einen Compiler von der imperativen Programmiersprache Rust in die pur funktionale Sprache des Theorembeweisers Lean vor, der nicht nur generell das erste Werkzeug zur Verifikation von Rust-Programmen darstellt, sondern diese insbesondere auch mithilfe der von Lean bereitgestellten Standardwerkzeugen und -logik ermöglicht. Diese Transformation ist nur möglich durch spezielle Eigenschaften von allen validen Rust-Programmen, die die Veränderbarkeit von Werten auf begrenzte Geltungsbereiche einschränken und statisch durch Rusts Typsystem garantiert werden. Die Arbeit demonstriert den Einsatz des Compilers anhand der Verifikation von Realbeispielen und zeigt die Erweiterbarkeit des Projekts über reine Verifikation hinaus am Beispiel von asymptotischer Laufzeitanalyse auf.

Abstract

Imperative programming, and aliasing in particular, represents a major obstacle in formally reasoning about everyday code. By utilizing restrictions the imperative programming language Rust imposes on mutable aliasing, we present a scheme for transforming a substantial part of the Rust language into the purely functional language of the Lean theorem prover. We use this scheme to verify the correctness of real-world examples of Rust code without the need for special semantics or logics. We furthermore show the extensibility of our transformation by incorporating an analysis of asymptotic runtimes.

Contents

1 Introduction **1**

2 Related Work **3**

3 Background **5**
3.1 Rust 5
3.2 Lean 9

4 Basic Transformation **10**

5 Case Study: Verification of `std::[T]::binary_search` **10**

6 Transformation of Mutable References **10**

7 Case Study: Verification of `fixedbitset` **10**

8 Conclusion and Future Work **10**

1 Introduction

Imperative programming languages are ubiquitous in today's software development, making them prime targets for formal reasoning. Unfortunately, their semantics differ from those of mathematics and logic – the languages of formal methods – in some significant details, starting with the very concept of "variables". The problem of mutability is only exacerbated for languages that allow references to *alias*, or point to the same memory location, enabling non-local mutation.

The standard way of verifying programs in such languages with the help of an interactive theorem prover is to explicitly model the semantics of the language in the language of the theorem prover, then translate the program to this representation (a "deep" embedding) and finally prove the correctness of its formalized behavior. This general approach is very flexible and allows for the verification of meta programs such as program transformations. The downside of the approach is that the theorem prover's tools and tactics may not be directly applicable to the embedded language, defeating many amenities of modern theorem provers. Alternatively, programs can be "shallowly" embedded by directly translating them into terms in the theorem prover's language without the use of an explicit inner semantics. This simplifies many semantic details such as the identification and substitution of bound variables, but is harder to accomplish the more the semantics of the source language differs from the theorem prover's own semantics.

verification condition generators? Move to Related Work?

Regardless of the type of embedding, an explicit heap that references can point into must generally be modeled and passed around in order to deal with the aliasing problem. References in this model may be as simple as indices into a uniform heap, but various logics such as separation logic [12] have been developed to work on a more abstract representation and to express aliasing-free sets of references.

Languages with more restricted forms of aliasing exist, however. Rust [10], a new, imperative systems programming language, imposes on mutable references the restriction of never being aliased by any other reference, mutable or immutable. This restriction eliminates the possibility of data races and other common bugs created by the presence of mutable sharing such as iterator invalidation. It furthermore enables more aggressive optimizations.

While the full Rust language also provides raw pointers, which are not bound by the aliasing restriction, and other "unsafe" operations, a (informal or formal) memory model for Rust has yet to be proposed. We therefore focus on the "safe" subset of Rust that has no unsolved semantic details.

We utilize safe Rust's aliasing restriction to design a monadic shallow embedding of a substantial subset of Rust into the purely functional language

of the Lean [4] theorem prover without the need for any heap-like indirection. This allows us to reason about unannotated, real-world Rust code in mostly the same manner one would reason about native Lean definitions. The monadic approach gives us further flexibility in modeling additional effects such as function runtime.

We first discuss the simpler cases of the translation, notably excluding mutable references, in Section 4 and show their application by giving a formal verification of Rust's `std::[T]::binary_search` method in Section 5. Section 6 discusses the translation of most usages of mutable references, which is used in Section 7 for a verification of the `fixedbitset` crate.

2 Related Work

While this thesis presents the first verification tool for Rust programs, tools for many other imperative languages have been developed before.

The Why3 project [2] is notable for its generality. It provides an imperative ML-like language *WhyML* together with a verification condition generator that can interface with a multitude of both automatic and interactive theorem provers. While WhyML supports advanced language features such as type polymorphism and exceptions, it does not support higher-order functions, which are ubiquitous in Rust code. WhyML provides a reference type `ref` that can point to a fresh cell on the heap and is statically checked not to alias with other `ref` instances, but cannot point into some existing datum like Rust references can. For example, the first of the following two WhyML functions fails to type check because the array elements are not known to be alias-free, while the second one will return a reference to a *copy* of `a[i]`.

```
let get_mut (a : array (ref int)) (i : int) : ref int = a[i]
let get_mut (a : array int) (i : int) : ref int = ref a[i]
```

WhyML is also being used as an intermediate language for the verification of programs in Ada [8], C [3] and Java [5]. For the latter two languages, aliasing is reintroduced by way of an explicit heap.

The remarkable SeL4 project [9] delivers a full formal verification of an operating system microkernel by way of multiple levels of program verification and refinement steps. The C code that produces the final kernel binary is verified by embedding it into the theorem prover Isabelle/HOL [11], using a deep embedding for statements and a shallow one for expressions. The C memory model used allows type-unsafe operations by use of a byte-size heap, but as with Why3, higher-order functions are not supported. The AutoCorres [6, 7] tool then transforms this representation into a shallow monadic embedding, dealing with the 'uninteresting complexities of C' [7] on the way. The result is an abstracted representation that is quite similar to ours (and in fact inspired it in part, as we shall note below), but doesn't go the last mile of completely eliminating the heap where possible. Without explicit no-alias annotations, the semantics of C would allow that in far fewer places than those of Rust in any case.

It should be noted that our work, like most verification projects based on either embedding or code extraction, relies on both an unverified compiler and an unverified embedding tool, effectively making both part of the trusted computing base. SeL4 is a notable exception in this, providing (at lower optimization levels) a direct equivalence proof [13] between the produced

kernel binary and the verified embedded code, thus completely removing the original C code from the trusted computing base.

```
struct Point { x: u32, y: u32 }
enum Option<T> { None, Some(T) }

fn map<S, T, F: Fn(S) -> T>(opt: Option<S>, f: F) -> Option<T> {
    match opt {
        Option::None    => Option::None,
        Option::Some(s) => Option::Some(f(s)),
    }
}
```

Figure 1: A first example of functional programming in Rust, showing algebraic data types, polymorphic and higher-order functions, pattern matching, type inference and the expression-oriented syntax

3 Background

We start by giving a basic introduction to our source and target languages, focusing on the parts relevant to our work. We will discuss finer semantic details where needed in Section 4 and Section 6.

3.1 Rust

Rust [10] is a modern, multi-paradigm systems programming language sponsored by Mozilla Research and developed as an open-source community effort. Rust is still a quite young language, with its first stable version having been released on May 15, 2015. The two biggest Rust project as of this writing are the Servo[1] [1] web browser engine and the Rust compiler `rustc`[2] itself.

As a partly functional language, Rust is primarily inspired by ML and shares much of its syntax, as evidenced in Figure 1. However, the syntax also shows influences by C, the dominant systems programming language of the present. Finally, Rust also features a *trait* system modeled after Haskell's type classes.

Many features of Rust other than the syntax can be explained by Rust's desire to feature an ML-like abstraction level while still running as efficiently as C, even on resource-constrained systems that may not allow dynamic allocation at all. Most prominently, Rust uses manual memory management just like C and C++, but guarantees memory safety through its *ownership* and *borrowing* systems. Rust also features an *unsafe* language subset that

[1] `https://github.com/servo/servo`
[2] `https://github.com/rust-lang/rust`

allows everything-goes programming on the level of C, but which is usually reserved for implementing low-level primitives on which the *safe* part of the language can then build. In general, safe Rust is (thought to be) a type-safe language like ML and unlike either C or C++. We focus on safe Rust in the following and in our work in order to peruse these guarantees.

Ownership describes the application of *linear types* to memory management as proposed by Wadler [14]. The owner of a Rust object is the binding that is responsible for freeing the object's resources (by calling a method of the `Drop` trait), which generally happens at the end of the binding's scope. Because an object managing resources should only ever have one owner, types that implement `Drop` are linear types: A value may only be used once, at which point it is consumed and ownership is transferred to its new binding[3]. In the following example, we extract an element from a `Vec` (a dynamically-sized array type that has to free heap space in its `Drop` implementation), after which we are not permitted to use the `Vec` again.

```
fn get<T>(v: Vec<T>, idx: usize) -> T {
    v[idx]
    // v will be freed here
}

let v: Vec<u32> = vec![1];
let x = get(v, 0);
// get(v, 1); // error[E0382]: use of moved value: `v`
```

One way to retain access to the `Vec` would be to also return it from the function, regaining ownership. However, since `T` in general is an linear type too, `get` would have to remove the indexed element before returning the `Vec`.

A much better alternative is to use *references*, which provide standard pointer indirection. Because a reference does not take ownership of the pointee, creating it is also called *borrowing*.

```
fn get<T>(v: &Vec<T>, idx: usize) -> &T {
    &v[idx]
}

let v: Vec<u32> = vec![1];
let x = get(&v, 0); // x: &u32
```

[3]Technically, because leaking resources (i.e. not consuming the object at all) is a safe operation in Rust, such types are merely *affine*. However, the distinction is not relevant for our purposes.

Here `&T` represents an immutable reference to a value of type `T`. Note that the compiler would stop us if we tried to return `v[idx]` by value:

```
error[E0507]: cannot move out of indexed content
```

Still, coming from other languages with manual memory management, this might look like a potentially unsafe thing to do: The function signature does not tell the callee that the returned reference is only valid as long as the `Vec`. Even Wadler tells us that a temporary reference to a linear value must be checked not to escape from the local scope. Indeed, it seems like the following program should produce a dangling pointer.

```
fn dangling() -> u32 {
    let x = {
        let v: Vec<u32> = vec![1];
        get(&v, 0)
        // v will be freed here
    };
    *x
}
```

However, the Rust compiler will stop us from doing this, printing an elaborate error message:

```
error: `v` does not live long enough
|
|         get(&v, 0)
|              ^ does not live long enough
|     };
|     - borrowed value only lives until here
|     *x
| }
| - borrowed value needs to live until here
```

The compiler must have had some information about the relationship of `x` and `v` in order to deduce this without resorting to inter-procedural analysis. It turns out that the full signature of the `get` function is as follows:

```
fn get<'a, T>(v: &'a Vec<T>, idx: usize) -> &'a T
```

'a is called a *formal lifetime parameter*. It specifies that the returned reference is indeed only valid as long as the first argument. By integrating lifetimes into the type system like this, Rust can reason about references even when confronted with complex, inter-procedural, higher-order reference lifetime relations.

While we have solved the dangling pointer problem for immutable data, mutability as so often aggravates the problem.

```
fn dangling2() -> u32 {
    let mut v: Vec<u32> = vec![1];
    let x = get(&v, 0);
    // remove all elements from v
    v.clear(); // shorthand for (&mut v).clear();
    *x
    // v will be freed here
}
```

By clearing the vector while we still hold a reference to its content, we should again produce a dangling pointer – even though this time, v indeed outlives x. Fortunately, the Rust compiler will again stop us:

```
error[E0502]: cannot borrow `v` as mutable because it is also
↪  borrowed as immutable
|
|     let x = get(&v, 0);
|                 - immutable borrow occurs here
|     // remove all elements from v
|     v.clear(); // shorthand for (&mut v).clear();
|     ^ mutable borrow occurs here
|     *x
| }
| - immutable borrow ends here
```

We have finally arrived at the aliasing problem: In a language with manual memory management, we can create type unsafety through the mere existence of two pointers, at least one of them mutable, to the same datum. Thus, Rust detects and forbids any occurrences of mutable aliasing, as shown above.

The beauty of forbidding mutable aliasing is that it solves many sources of bugs in imperative programs even outside of managed memory management. Indeed, as Wadler notes, it makes mutable references safe even in

a referentially transparent language: “In order for destructive updating of a value to be safe, it is essential that there be only one reference to the value when the update occurs” [14]. While Rust does introduce APIs such as for I/O that break referential transparency, the absence of mutable aliasing still provides safety guarantees that are usually only attributed to purely functional languages, first and foremost among them the elimination of data races. By focusing on a subset of Rust and its APIs that is truly referentially transparent, we obtain a sufficiently narrow gap between Rust and the purely functional language Lean that our transformation between them becomes feasible.

3.2 Lean

4 Basic Transformation

5 Case Study: Verification of `std::[T]::binary_search`

6 Transformation of Mutable References

7 Case Study: Verification of `fixedbitset`

8 Conclusion and Future Work

I have no idea where to put the complexity analysis

References

[1] 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*, pages 81–89. ACM, 2016.

[2] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[3] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[4] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[5] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *International Conference on Computer Aided Verification*, pages 173–177. Springer, 2007.

[6] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *International Conference on Interactive Theorem Proving*, pages 99–115. Springer, 2012.

[7] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: formal verification of C code without the pain. *ACM SIGPLAN Notices*, 49(6):429–439, 2014.

[8] J. Guitton, J. Kanig, and Y. Moy. Why Hi-Lite Ada. *Rustan, et al.[32]*, pages 27–39, 2011.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[10] N. D. Matsakis and F. S. Klock II. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[12] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[13] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482, New York, NY, USA, 2013. ACM.

[14] P. Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

Erklärung

Hiermit erkläre ich, Sebastian Andreas Ullrich, dass ich die vorliegende Masterarbeit 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