Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

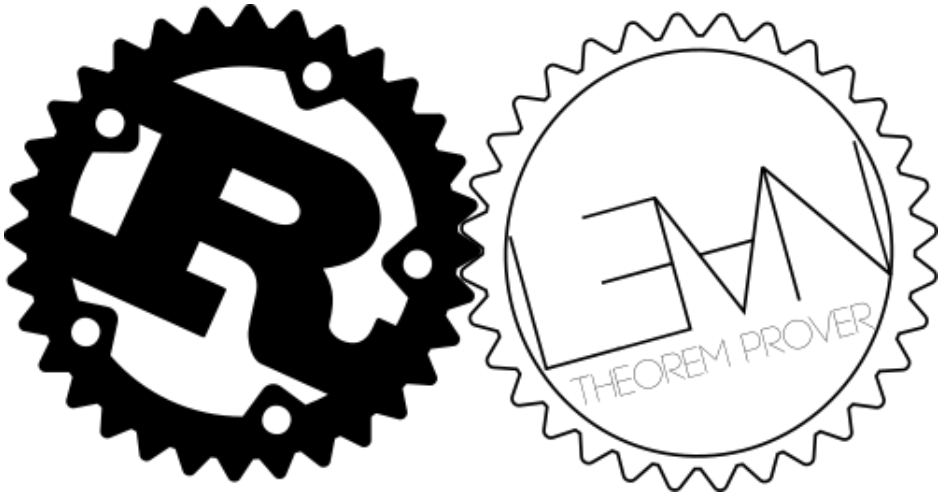
Lehrstuhl Prof. Dr.-Ing. Snelting

# 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 shallowly embedding 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

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 it is harder to accomplish the more the semantics of the source language differs from the theorem prover's own semantics.

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 [16] 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 [12], 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 memory model for Rust (informal or formal) 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 [6] 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. We 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 [3] 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]
```

In contrast, Rust can provide a perfectly safe function with this functionality.

```
fn slice::get_mut<T>(slice: &mut [T], index: usize) -> Option<&mut T>
```

WhyML is also being used as an intermediate language for the verification of programs in Ada [10], C [5] and Java [7]. For the latter two languages, aliasing is reintroduced by way of an explicit heap.

The remarkable SeL4 project [11] 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 [13], 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 [8, 9] tool then transforms this representation into a shallow monadic embedding, dealing with the 'uninteresting complexities of C' [9] 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. Thus the user still has to worry and reason about (the absence of) aliasing manually or through a nested logic such as separation logic. Without explicit no-alias annotations, the semantics of C would allow eliminating the heap 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 [17] between the produced kernel binary and the verified embedded code, thus completely removing the original C code from the trusted computing base.

While not an imperative language, the purely functional, total Cogent language [14] uses linear types in the style of Wadler [18] for safe manual memory management, much like Rust. The language is designed both to be easily verifiable (by building on AutoCorres) and to compile down to efficient C code. As we shall see in Subsection 3.1, the biggest differences between Wadler-style purely functional linear languages and Rust are the existence of mutable references as well as sophisticated interprocedural reference tracking in the latter. For example, the aforementioned `get_mut` function can only be expressed as a higher-order function in Cogent, even in the immutable case.

# 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 [12] 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 projects 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 Listing 1. However, the syntax also shows influences by C, the dominant systems programming language at 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 allows everything-goes programming on the level of C, but which is usually

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

```
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)),
    }
}
```

**Listing 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

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 [18]. 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 a 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
```

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
```

[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.

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" [18]. 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

The Lean [6] theorem prover is an open source, dependently typed, interactive theorem prover developed jointly at Microsoft Research and Carnegie Mellon

University. The first official release of Lean was announced at CADE-25 in August 2015, making it just a few months younger than Rust. As of this writing, development on Lean is focused on the next, unreleased version that will feature powerful automation written in Lean itself.

Lean supports two different interpretations of Martin-Löf type theory: a purely constructive one based on Homotopy Type Theory, and one based on the Calculus of Inductive Constructions [4, 15] as championed by the Coq [2] theorem prover, which supports both constructive and classical reasoning. We use the latter in our work.

The primitive type in dependent type theory is the dependent function (or product) type $\Pi x : A, B$, where $x$ may occur in $B$; if it does not, we obtain the standard function type $A \to B$. Function abstraction and application extend naturally to dependent functions, as perhaps best described by their formal typing rules.

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A, e) : \Pi x : A, B} \qquad \frac{\Gamma \vdash f : \Pi x : A, B \qquad \Gamma \vdash e : A}{\Gamma \vdash fe : [e/x]B}$$

The Calculus of Inductive Constructions extends basic dependent type theory with a type scheme for inductive types, which are described by a set of (possibly dependent) functions, their *constructors*. `??` shows basic inductive definitions from the Lean standard library.

As we can see in Listing 2, inductive types themselves are instances of a type, namely `Type`. This turns out to be a slight simplification, however. More specifically, Lean has a whole hierarchy of indexed types or *universes* `Type.{i}`, with `Type.{i} : Type.{i+1}`[4]. The universe hierarchy is needed to avoid the type theoretic equivalent of Russell's paradox. When we write just `Type` for the type of an inductive definition like in `??`, a correct universe level $i \geq 1$ (possibly dependent on argument universe levels) will automatically be inferred. The reason for skipping `Type.{0}` is that it has a special function that is suggested by its more common name, `Prop`: It is the universe we normally declare types in that are to be interpreted as propositions.

Under the Curry-Howard isomorphism, an objects of a type can be interpreted as a proof of a proposition. The reason Lean uses a separate universe for this interpretation is that `Prop` is given a specific property that would not make sense for the other universes: By *proof irrelevance*, any two objects of a type in `Prop` are definitionally equal. In other words, proofs are irrelevant for computation. Finally, `Prop` is also *impredicative*: If `B : Prop`, then also

[4]The hierarchy is, however, not cumulative: It is not true that `Type.{i} : Type.{i+2}`.

```
inductive empty : Type

inductive unit : Type :=
star : unit

inductive prod (A B : Type) : Type :=
mk : A → B → prod A B

inductive sum (A B : Type) : Type :=
| inl : A → sum A B
| inr : B → sum A B

-- the dependent sum type
inductive sigma (A : Type) (B : A → Type) : Type :=
mk : Π(x : A), B x → sigma A B

inductive bool : Type :=
| ff : bool
| tt : bool

inductive option (A : Type) : Type :=
| none : option A
| some : A → option A

inductive nat : Type :=
| zero : nat
| succ : nat → nat
```

**Listing 2:** The most basic inductive types as well as some basic types from functional programming in Lean

| type name (notation) | |
|---|---|
| in **Type** | in **Prop** |
| `empty` | `false` |
| `unit` | `true` |
| `prod` (`×`) | `and` (`∧`) |
| `sum` (`+`) | `or` (`∨`) |
| `sigma` (`Σ x : A, B`) | `Exists` (`∃ x : A, B`) |
| `Π x : A, B` | `∀ x : A, B` |
| `A → B` | `A → B` |

**Table 1:** The basic Curry-Howard correspondence. The table lists types from Listing 2 and the corresponding types from the standard library with the same constructors, but declared in **Prop**. We also show their notations as well as the special universal quantifier notation for dependent functions into **Prop**. Nondependent functions and implications are not distinguished by notation.

`(Π x : A, B) : Prop` for any `A`. This property ensures that predicates and universal quantifications are still propositions. The separation of inductive types and inductive propositions can lead to some duplication, which however turns out to be very useful in ensuring suggestive names and notations for each side (Table 1).

On top of its interpretation of dependent type theory, Lean includes many notational amenities. On the type level, in addition to basic and inductive definitions, it features syntactic **abbreviation**s as well as **structure**s. The latter are single-constructor inductive types that automatically define projections to each of their constructor parameters (or *fields*) and furthermore support inheriting fields from other structures.

```
structure point2 :=
(x : ℕ)
(y : ℕ)

structure point3 extends point2 :=
(z : ℕ)

example : point2 := {point2, x := 0, y := 1}
check point3.x -- point3.x : point3 → ℕ
```

In addition to the standard parameter syntax `(x : A)`, Lean also supports two more binding modes, `{x : A}` and `[x : A]`. In the first one, `x` is an *implicit* parameter and will be inferred from other parameters or the expected

result type, such as in the constructor of the ubiquitous type `eq` modeling Leibniz equality:

```
inductive eq {A : Type} (a : A) : A → Prop :=
refl : eq a a -- explicit form: @eq A a a
```

The binding mode `[x : A]` instructs Lean to infer `x` by *type class inference*. Type classes are arbitrary definitions annotated with the `[class]` attribute. Type class inference synthesizes instances of a class by a Prolog-like search through definitions of the class type marked with `[instance]`.

```
structure inhabited [class] (A : Type) : Type :=
(value : A)

definition default (A : Type) [inhabited A] : A :=
inhabited.value A

definition nat.is_inhabited [instance] : inhabited ℕ :=
⟦inhabited, value := 0⟧
definition prod.is_inhabited [instance] (A B : Type)
  [inhabited A] [inhabited B] : inhabited (A × B) :=
⟦inhabited, value := (default A, default B)⟧

eval default (ℕ × ℕ) -- (0, 0)
```

In order to keep definition signatures short, we will also make use of Lean's `section` mechanism that allows us to fix common parameters for a set of definitions.

```
section
  -- in this section, implicit in signatures and in use sites
  parameter (A : Type)
  -- implicit in signatures but explicit in use sites
  variable (x : A)

  definition f : A := x
  check f -- f : A → A
end

check f -- f :  A : Type, A → A
```

extend ad nauseam when needed

# 4 Basic Transformation

In this section, we describe the basic translation from Rust to Lean that includes pure code as well as mutable local variables and loops, but not mutable references (see Section 6). We roughly follow the structure of the Rust Reference[5].

## 4.1 The MIR

Because Rust makes extensive use of inference algorithms for types, lifetimes and typeclasses, correctly parsing Rust code is no small feat. Therefore, we use the Rust Compiler `rustc` itself as a frontend and work on the completely explicit and much simpler *mid-level intermediate representation* (MIR) (Figure 1). By writing our translation program in Rust, we can import the `rustc` libraries to gain access to the MIR and many convenient helper functions.

The MIR is a control flow graph (CFG) representation where a basic block consists of a list of statements followed by a terminator that (conditionally or unconditionally) transfers control to other basic blocks. For readability, this section will mostly argue on the Rust source level, but the graph structure will be important for translating control flow.

[5] `https://doc.rust-lang.org/reference.html`

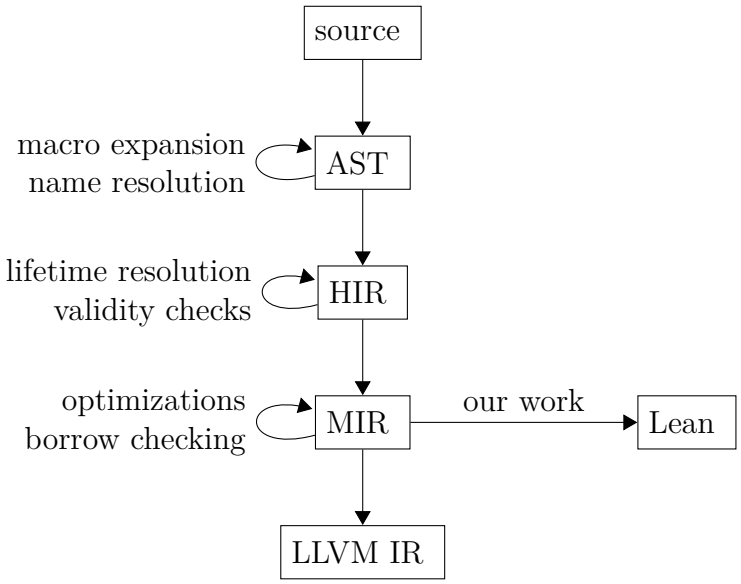



**Figure 1:** Overview of the Rust compiler pipeline and our work in that context

## 4.2 Programs and files

Rust's unit of compilation is called a *crate*. A crate consists of one or more `.rs` files and can be compiled to an executable or library. Files inside a crate may freely reference declarations between them. On the other hand, Lean files may only import other files non-recursively and declarations must be strictly sorted in order of usage for termination checking. We therefore translate a crate into a single Lean file and perform a topological sort on its declarations. While Lean does support explicit declarations of mutually recursive types and functions, we have not yet encountered such declarations in Rust code as part of our formalization work and thus have not implemented support for them so far.

In detail, our tool creates a file called `generated.lean` in a separate folder for each crate and connects them using Lean's `import` directive according to the inter-crate dependencies. The user can additionally create a `pre.lean` file that will automatically be imported and can be used for axiomatizations as well as a `config.toml` file that can influence the translation – see below for examples. We use a third Lean file `thy.lean` per crate for the proofs, which will import both the generated code and proof files from other crates.

## 4.3 Types

### 4.3.1 Primitive Types

Rust's primitive types are the boolean type, machine-independent and machine-dependent integer types, tuples, arrays, slices, and function types.

Following AutoCorres's design, we map the primitive integer types to Lean's native arbitrary-sized types and instead handle overflow explicitly during computation (Subsection 4.6.1).

```
abbreviation u8 [parsing_only] := nat
abbreviation u16 [parsing_only] := nat
abbreviation u32 [parsing_only] := nat
abbreviation u64 [parsing_only] := nat
abbreviation usize [parsing_only] := nat

abbreviation i8 [parsing_only] := int
// ...

definition u8.bits [reducible] : ℕ := 8
// ...

definition usize.bits : ℕ := 16
lemma usize.bits_ge_16 : usize.bits ≥ 16 := dec_trivial
attribute usize.bits [irreducible]
```

For the machine-size integer types **usize** and **isize**, we only expose the conservative assumption that their bit sizes are at least 16. We still define **usize.**bits to be exactly 16 so that it is computable, but by then marking the definition as [irreducible], this fact is only accessible in proofs when explicitly unfolding the definition. When a proof does rely on the bounds of a parameter, we can add a separate hypothesis, for which we make use of typeclasses. The bounds of an expression can often be determined just from partial information, such as with unsigned division.

```
definition is_bounded_nat [class] (bits x : ℕ) := x < 2^bits
abbreviation is_usize := is_bounded_nat usize.bits

lemma div_is_bounded_nat [instance] (bits x y : ℕ)
  [is_bounded_nat bits x] : is_bounded_nat bits (x / y) := ...
```

We use the same approach for arrays ([T; N]) and slices (&[T]), mapping both the to arbitrary-length list type. While Rust arrays have a constant length encoded in the type, slices are dynamic views into contiguous sequences like arrays or Vecs and bounded only by the memory size. The latter fact is important when trying to prove that a **usize** counter can reach all indices in a slice.

```
abbreviation array [parsing_only] (A : Type₁) (n : ℕ) := list A
abbreviation slice [parsing_only] := list

definition is_slice [class] {T : Type₁} (xs : slice T) :=
length xs ≤ 2^usize.bits
```

### 4.3.2 Structs and enums

Because Rust does not feature inheritance, struct types and enumerated types are true Algebraic Data Types and can directly be translated to their Lean equivalents (**structure** and **inductive**, respectively).

### 4.3.3 References

An immutable reference &'a T is checked by the Rust compiler not to alias with any mutable reference and thus can be safely replaced with the translation of T itself. We drop all lifetime specifiers in general because we trust the Rust compiler to already have made the memory safety checks.

We will discuss mutable references in Section 6.

## 4.4 The semantics monad

The core part for representing Rust's dynamic semantics is the monadic embedding. While higher-order unification issues in the current Lean version prevent us from outright parameterizing the embedding by an arbitrary monad instance, we still try to keep the interface of our specific monad abstract so that the monad can be extended in the future.

We currently model abnormal termination[6] and nontermination as well as an abstract step counter for asymptotic run time analysis.

```
definition sem (A : Type₁) := option (A × ℕ)
```

We provide the standard monadic operations on the type, including a `do`-notation. The model-specific operations are `mzero` indicating abnormal termination/nontermination, and `sem.incr`, which increments the step counter (if any). An increment of one is emitted around every Rust function call and before each loop iteration.

```
definition mzero {A : Type₁} : sem A := none
definition return {A : Type₁} (x : A) : sem A := some (x, 0)

definition sem.incr {A : Type₁} (n : ℕ) : sem A → sem A
| (some (x, k)) := some (x, k+n)
| none          := none

definition sem.bind {A B : Type₁} (m : sem A) (f : A → sem B)
  : sem B :=
option.bind m (λs, match s with
| (x, k) := sem.incr k (f x)
end)

infixl ` >>= `:2 := sem.bind
```

The semantics monad follows the usual monad laws, which we will make use of in proofs.

```
lemma return_bind {A B : Type₁} {a : A} {f : A → sem B}
  : (return a >>= f) = f a := ...
lemma bind_return {A : Type₁} {m : sem A} : (m >>= return) = m := ...
lemma bind.assoc {A B C : Type₁} {m : sem A} {f : A → sem B}
  {g : B → sem C} : (m >>= f >>= g) = (m >>= (λx, f x >>= g)) := ...
```

[6] unspecified behavior like integer overflow and *panics* from out-of-bounds array accesses or explicit `panic!` calls. Rust does not have exceptions.

## 4.5 Statements and control flow

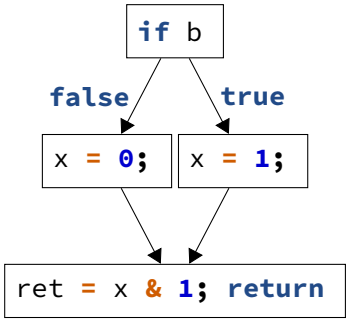
The local state of a Rust function consists of its arguments, variables, and temporaries (variables introduced by the compiler). Without mutable references, these locals can only be manipulated by assignments, the single statement kind available in the MIR. In linear code, keeping track of assignments is as easy as transforming them to redeclarations.

```
p.x += 1;
```

```
let p = Point { x = p.x + 1, ..p };
```

Nonlinear control flow is introduced by Rust's `if` and `match` constructs as well as its three loop constructs (which have a single common representation in the MIR). We map the first two cases to Lean's corresponding constructs of the same names.

```
let x = if b {1} else {0};
x & 1
```




```
if b = bool.tt then
  let x := 1 in
  x & 1
else
  let x := 0 in
  x & 1
```

As can be seen, we currently translate each branch of a conditional block terminator independently, which can lead to code duplication if those branches converge again. While this has not manifested any problems in our verification work so far, we may want to mitigate it in the future by factoring out the common translated code into a separate definition.

```
fn f() {
  let mut x = 0;
  while x < 10 {
      x += 1;
  }
}
```

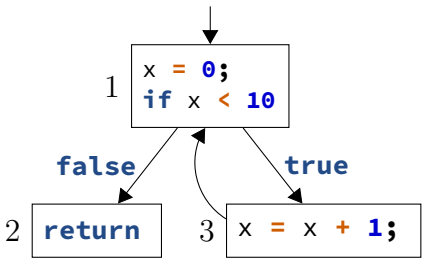



**Figure 2:** A `while` loop and the corresponding (simplified) MIR graph. Blocks 1 and 3 from a strongly connected component, which is dominated by block 1, the loop header.

We do need to factor out common code in the case of loops. There is no special terminator signifying loops in the MIR; instead, we have to search for (nontrivial) strongly connected components (SCCs) of basic blocks (Figure 2). Because Rust's control flow is *reducible* (notably, lacking a *goto* instruction), we may assume that such an SCC can only be entered from a single node (*dominating* the SCC). With this, we can describe the semantics of the SCC in more traditional terms of iteration: The dominating node is the *loop header*, while the rest of the SCC is the *body*. Jumping back to the header signifies a new iteration, while jumping out of the SCC means breaking the loop. By breaking up the SCC at the header, we can thus translate a single iteration to a function of type

```
State → sem (State + Res)
```

that takes a tuple `State` of loop variables and either returns the new state for the next iteration, or a value of the source function's return type `Res` when breaking out of the loop. We tie this into a single value of type `sem Res` by use of a general *loop combinator*.

### 4.5.1 The loop combinator

The loop combinator has the signature

```
noncomputable definition loop {State Res : Type₁}
  (body : State → sem (State + Res)) (s : State) : sem Res
```

Its task is to apply `body` repeatedly (starting with `s`) until some `Res` is returned; if the loop does not terminate, it returns `mzero` (which `body` may also return by itself). Termination for arbitrary values of `body` obviously is not a decidable property. Therefore we will have to leave the constructive subset of Lean, as signified by the **`noncomputable`** specifier. The (simplified) translation of the Rust code in Figure 2 via `loop` is as follows:

```
definition f.loop_1 (x : i32) : sem (i32 + unit) :=
if x < 10 then
  let x := x + 1 in
  return (sum.inl x)
else
  return (sum.inr unit.star)

definition f : sem unit :=
let x := 0 in
loop f.loop_1 x
```

As a total, purely functional language, Lean cannot express iteration directly, and the only primitive kind of recursion available in Lean is structural recursion over an inductive datatype. On top of structural recursion, the Lean standard library defines the more general concept of *well-founded* recursion: A relation `R : A → A → Prop` on a type `A` is well-founded if every element of `A` is *accessible* through the relation, which is defined inductively as all predecessors of the element under the relation being accessible.

```
inductive acc {A : Type} (R : A → A → Prop) : A → Prop :=
intro : ∀x, (∀ y, R y x → acc R y) → acc R x

inductive well_founded [class] {A : Type} (R : A → A → Prop) : Prop
↪  :=
intro : (∀ a, acc R a) → well_founded R
```

Using structural recursion over the `acc` predicate, the standard library defines a fixed-point combinator for functionals respecting a well-founded relation, and proves that the combinator satisfies the fixpoint equation.

```
namespace well_founded
section
  variables {A : Type} {C : A → Type} {R : A → A → Prop}

  definition fix [well_founded R] (F : Πx, (Πy, R y x → C y) → C x)
    (x : A) : C x := ...

  theorem fix_eq [well_founded R] (F : Πx, (Πy, R y x → C y) → C x)
    (x : A) : fix F x = F x (λy h, fix F y) := ...
end
end well_founded
```

We use well-founded recursion to define `loop`: If repeatedly applying *body* to $s$ yields a sequence of states, this sequence will terminate iff there exists a well-founded relation on `State` such that the sequence is a descending chain. This is true because descending chains in well-founded relations are finite, and conversely a finite sequence $s_1 = s, \ldots, s_n$ is a descending chain in the trivial well-founded relation $R = \{(s_{i+1}, s_i) | 1 \leq i < n\}$.

In the formalization, given a well-founded relation `R` on `State`, we first have to take care of lifting it to a well-founded relation `R'` on `State + Res`.

```
section
  parameters {State Res : Type₁}
  parameter (body : State → sem (State + Res))
  parameter (R : State → State → Prop)

  definition State' := State + Res
```

```
  definition R' : State' → State' → Prop
  | (inl s') (inl s) := R s' s
  | _        _       := false

  private lemma R'.wf [instance] [well_founded R] : well_founded R'
↪  := ...
```

We can then wrap `body` in a functional respecting `R'` that we can pass to `well_founded.fix`.

```
  definition F (x : State') (f : Π (x' : State'), R' x' x → sem
↪  State') : sem State' :=
  match x with
  | inr _ := mzero -- unreachable
  | inl s :=
    do x' ← sem.incr 1 (body s);
    match x' with
    | inr r := return (inr r)
    | x'    := if H : R' x' x then f x' H else mzero
    end
  end

  definition loop.fix [well_founded R] (s : State) : sem Res :=
  do x ← well_founded.fix F (inl s);
  match x with
  | inr r := return r
  | inl _ := mzero -- unreachable
  end
```

Finally, we implement `loop` by choosing any well-founded relation `R` that makes the loop terminate, if any, or else return `mzero`.

```
  definition terminating (s : State) :=
  ∃ Hwf : well_founded R, loop.fix s ≠ mzero

  noncomputable definition loop (s : State) : sem Res :=
  if Hex : ∃ R, terminating R s then
    @loop.fix (classical.some Hex) _ (classical.some
↪  (classical.some_spec Hex)) s
  else mzero
```

Here we make use of the *dependent if-then-else* notation that allows us to test for a property and then bind a name to a proof of it in case it holds. We then destructure that proof object to obtain the relation and its well-foundedness proof so that we can pass them to `loop.fix`. The `classical.some` and `classical.some_spec` definitions are based on Hilbert's epsilon operator.

```
noncomputable definition classical.some {A : Type} {P : A → Prop} (H
↪  : ∃x, P x) : A := ...
theorem classical.some_spec {A : Type} {P : A → Prop} (H : ∃x, P x) :
↪  P (some H) := ...
```

The use of `classical.some` as well as the undecidable conditional `∃ R, terminating R s` make `loop` non-computable.

When verifying loops, we will first verify the corresponding application of `loop.fix` using a specific well-founded relation, for which we can prove a convenient fixpoint equation.

```
theorem loop.fix_eq
  {R : State → State → Prop} [well_founded R] {s : State} :
  loop.fix R s =
    do x' ← sem.incr 1 (body s);
    match x' with
    | inl s' := if R s' s then loop.fix R s' else mzero
    | inr r  := return r
    end := ...
```

If the application of `loop.fix` terminates, we can show that the original application `loop` will do so too with the same return value, via a helper lemma that says that all terminating `loop.fix` applications are equal.

```
lemma loop.fix_eq_fix
  {R₁ R₂ : State → State → Prop} [well_founded R₁] [well_founded R₂]
  {s : State}
  (Hterm₁ : loop.fix R₁ s ≠ mzero)
  (Hterm₂ : loop.fix R₂ s ≠ mzero) :
  loop.fix R₁ s = loop.fix R₂ s := ...

theorem loop.fix_eq_loop
  {R : State → State → Prop} [well_founded R]
  {s : State}
  (Hterm : loop.fix R s ≠ mzero) :
  loop.fix R s = loop s :=
```

## 4.6 Expressions

### 4.6.1 Arithmetic expressions

# 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] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[3] 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.

[4] T. Coquand and G. Huet. The calculus of constructions. *Information and computation*, 76(2-3):95–120, 1988.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. Murray, Y. Nagashima, T. Sewell, and G. Klein. Refinement through restraint: bringing down the cost of verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 89–102. ACM, 2016.

[15] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

[16] 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.

[17] 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.

[18] 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 |