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 [20] 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 [16], 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 [7] 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 `[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 partial 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 get_mut<T>(slice: &mut [T], index: usize) -> &mut T
```

WhyML is also being used as an intermediate language for the verification of programs in Ada [12], C [6] and Java [8]. For the latter two languages, aliasing is reintroduced by way of an explicit heap.

The remarkable SeL4 project [15] 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 [17], 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 [10, 11] tool then transforms this representation into a shallow monadic embedding, dealing with the 'uninteresting complexities of C' [11] 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 [21] 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 [18] uses linear types in the style of Wadler [23] 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 [16] 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 reserved for implementing low-level primitives on which the *safe* part of the language can then build. In general, safe Rust is

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

```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 (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 [23]. 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" [23]. 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 [7] 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 [5, 19] 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.

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

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*. Listing 2 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 Listing 2, 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 `(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

[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.

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 The 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 focus on the parts that are unique to Rust or are nontrivial to translate. We roughly follow the structure of the Rust Reference.[5] Because our translation output is not optimized for readability, all sample translations in this section have been prettified manually without changing their semantics. An non-prettified feature-by-feature breakdown is also available online.[6]

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

[5] `https://doc.rust-lang.org/reference.html`
[6] http://kha.github.io/electrolysis/

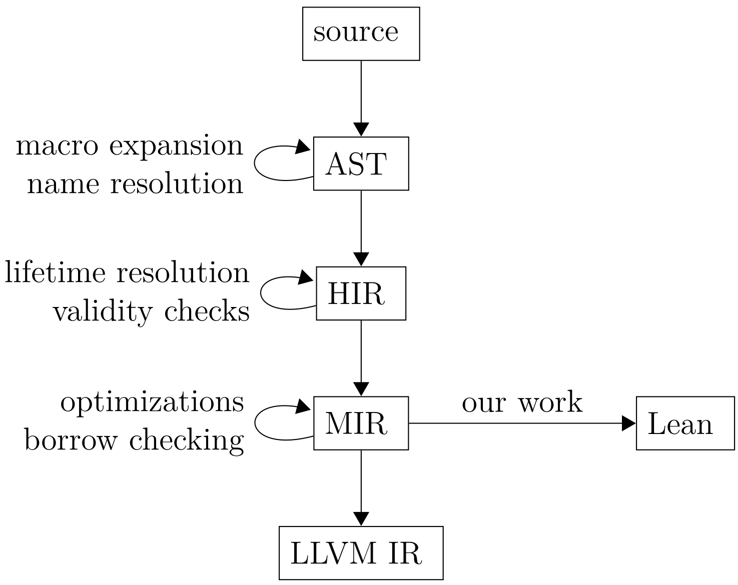


Figure 1: Overview of the Rust compiler pipeline and our work in that context

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.

4.2 Identifiers

Working on top of the MIR, we do not have to worry about the lexical structure of Rust. We do, however, have to make sure we emit lexically correct Lean code. This is only a problem with identifiers, which we would like to transfer with minimal changes. Both languages are based on segmented identifiers, just with different separators (`a::b::c` in Rust versus `a.b.c` in Lean). However, some identifier parts in Rust such as `[T]` or `<F<T> as S>` are not valid in Lean. To retain readability, we have therefore extended Lean with a general escaping syntax for identifiers that allows arbitrary symbols by surrounding them with `«` and `»`: The identifier `«[T]».«a.b»` is now a valid Lean identifier consisting of the parts `«[T]»` and `«a.b»`.

4.3 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.

4.4 Types

4.4.1 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' design (see Section 2), we map the primitive integer types to Lean's native arbitrary-sized types and instead handle overflow explicitly during computation (Subsection 4.8.1).

```
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 `Vec`s and bounded only by the memory size. More specifically, they (and any Rust type) are assumed to be no larger than `isize::MAX` bytes so that the pointer difference of any two elements can be represented by an `isize` value.

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

definition is_slice [class] {A : Type₁} (xs : slice A) :=
length xs < 2^(usize.bits-1)
```

4.4.2 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).

4.4.3 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 all memory safety checks.

We will discuss mutable references in Section 6.

4.5 Traits

Rust's trait system is similar to Haskell's type classes, but borrows some syntax from more object-oriented *interface* systems. In particular, in addition to functions a trait may also contain methods that can be called on any object of a type the trait is implemented *on*. This is implemented via an implicit type parameter `Self` that is used for the type of the `self` parameter and is specified in the `for` clause when implementing a trait via an `impl` block.

The translation of basic traits into Lean type classes is straightforward (Figure 2). We will discuss the details of the function-level translation and the `sem` monad below. While `impl` blocks in Rust are anonymous, we need to name all definitions in Lean and do so using a naming scheme similar to `rustc`'s own internal representation.

4.5.1 Default Methods

As shown in Figure 2, we generate separate definitions for functions in trait implementations before assembling them into a type class instance. This way, and by eliminating the type class indirection in calls to a statically known implementation, we can allow trait implementation functions to call each other using our standard topological dependency ordering.

```rust
struct S { i: i32 }

trait Trait<T> {
  fn f(self) -> T;
}

impl Trait<i32> for S {
  fn f(self) -> i32 {
    self.i
  }
}

fn g<T : Trait<i32>>
  (t: T) -> i32 {
  t.f()
}

fn h() -> i32 {
  g(S { i: 0 })
}
```

```
structure S := (i : i32)

structure Trait [class] (Self T : Type₁) :=
(f : Self → sem T)

definition «S as Trait<i32>».f (self : S) :
↪ sem i32 :=
return (S.i self)

definition «S as Trait<i32>» [instance] :=
⦃Trait S i32, f := «S as Trait<i32>».f⦄

definition g {T : Type₁} [Trait T i32] (t : T)
↪ : sem i32 :=
sem.incr 1 (Trait.f _ t)

definition h : sem i32 :=
sem.incr 1 (g (S.mk 0))
```

Figure 2: A parametric trait in Rust and its translation.

```rust
struct S;

trait Trait {
  fn f(self);
  fn g(self);
}

impl Trait for S {
  fn f(self) {
    self.g()
  }
  fn g(self) {}
}
```

```
structure S := (i : i32)

structure Trait [class] (Self : Type₁) :=
(f : Self → sem unit)
(g : Self → sem unit)

definition «S as Trait».g (self : S) : sem
↪ unit :=
return unit.star

definition «S as Trait».f (self : S) : sem
↪ unit :=
sem.incr 1 («S as Trait».g self)

definition «S as Trait» [instance] :=
⦃Trait S, f := «S as Trait».f, g := «S as
↪ Trait».g⦄
```

However, just like Haskell, Rust also allows default implementations of trait methods that may arbitrarily call and be called from other trait methods that will only be defined in some implementation of the trait later on. This makes static ordering of dependencies impossible.

In essence, a default method in a trait takes as input an instance of that trait to call other trait methods with, but at the same time has to be a slot in the very same trait because it may be overridden in an implementation.

There are multiple potential ways to deal with that depdendency cycle. We could simply create a specialized copy of the trait method for each instantiation, but then we would also have to copy proofs about it. We could try to dynamically solve the cycle in a general way, computing its least fixed point by use of the Knaster-Tarski theorem [22] as usual in denotational semantics. Or we can restrict ourselves to special cases that break the cycle. If we remove the trait instance as an input to the default method, it cannot call other trait methods. If, on the other hand, we do not make default methods part of the trait instance, it cannot be overridden or be called from inside implementations of the trait. We could even mix these two approaches, incrementally building up the trait instance by alternating between default and non-default methods.

We could implement all these approaches and automatically or manually choose between them on a case-by-case basis. It turns out, however, that in the Rust standard library, default methods are often just convenience wrappers around other trait methods, like in the `PartialEq` trait.

```
pub trait PartialEq<Rhs> {
  fn eq(&self, other: &Rhs) -> bool;
  fn ne(&self, other: &Rhs) -> bool { !self.eq(other) }
}
```

Therefore, as of now we have only implemented the third approach of declaring default methods outside of their trait, which turned out to be sufficient for our verification work so far.

```
structure PartialEq [class] (Self Rhs : Type₁) :=
(eq : Self → Rhs → sem bool)

definition PartialEq.ne {Self Rhs : Type₁} [PartialEq Self Rhs] (self :
↪  Self) (other : Rhs) : sem bool :=
do t5 ← sem.incr 1 (PartialEq.eq self other);
return (bool.bnot t5)
```

4.5.2 Associated Types

There is one further advanced trait feature Rust shares with Haskell called *associated types*: trait members that are not functions, but types.

```
pub trait Add<RHS> {
  type Output;
  fn add(self, rhs: RHS) -> Output;
}
```

Making `Output` an associated type instead of a type parameter fundamentally changes type class inference: Instead of being an input parameter to the inference like `Self` and `RHS`, `Output` is *determined* by the inferred trait instance. This means that inference on `add` can succeed even if the expected return type is unknown.

As a dependently typed language, Lean has no problem with representing such traits as type classes. What it cannot represent, however, is a special class of trait bounds Rust supports: `T : Add<RHS, Output=RHS>` asserts a definitional equality on the associated type; but definitional equality exists only as a judgment in Lean, not as a proposition we could pass as a parameter. Instead, we follow the original paper [4] on associated types in Haskell that translates type classes with associated types into System F by turning them into type parameters.

```
structure Add [class] (Self RHS Output : Type₁) :=
(add : Self → RHS → sem Output)
```

This transformation does weaken type class inference, which means that in the generated Lean code, we have to resort to passing type class arguments explicitly using the `@` notation. We might be able to regain inference in a potential future version of Lean that supports functional dependencies [14].

4.5.3 Trait Objects

Lastly, Rust's trait system exhibits a feature that does not directly exist in Haskell. In Haskell, type classes are not types - they cannot explicitly be passed by value, only implicitly through inference. In Rust, traits are dynamically sized types, which means they can be used as values, but only behind some indirection like `&Trait`. These *trait objects* are represented as a pointer to a vtable of the trait implementation and another pointer to the *Self* value.

This "fat pointer" representation would translate quite naturally to an existential type `Σ (Self : Type), (Trait Self × Self)`. What is not apparent in this natural definition, however, is the fact that it necessarily lives in a higher universe than `Self`. This is the only construct currently in Rust that can give rise to a type not in `Type₁` (but, in fact, to a type in an arbitrarily high universe through nesting). It is an open problem in the Lean community if and how a monad over types of different universes can cleanly work given Lean's non-cumulative universe hierarchy. Fortunately, trait objects are a rare feature in Rust code that we do not expect to find on the algorithmical level of our current verification work, so we have not investigated this issue any further for now.

4.6 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[7] 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)) := ...
```

4.7 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.

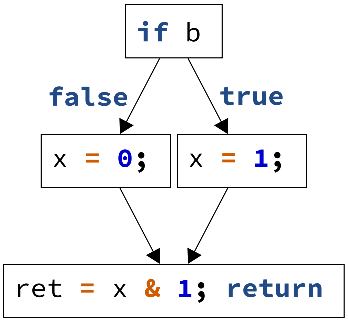
[7] unspecified behavior like integer overflow and *panics* from out-of-bounds array accesses or explicit `panic!` calls. Rust does not have exceptions.

```
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.

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 3). 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

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

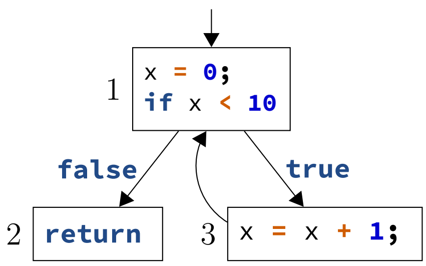


Figure 3: A `while` loop and the corresponding (simplified) MIR graph. Blocks 1 and 3 from a strongly connected component, which is dominated by block 1, the loop header.

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*.

4.7.1 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 3 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 := ...
```

4.8 Expressions

4.8.1 Arithmetic Operators

Rust's arithmetic semantics is based on the premise that in most circumstances, arithmetic overflow is unintended by the programmer,[8] and constitutes a bug in the program. Therefore, in debug builds, the built-in arithmetic operators will panic on any overflow. In release builds, overflows for both signed and unsigned types will wrap for performance reasons.

We thus regard arithmetic overflow in those operators as *unspecified* and return the bottom value in such cases, using the predicate `is_bounded_nat` from Subsection 4.4.1.

```
definition sem.guard {a : Type₁} (p : Prop) [decidable p] (s : sem a) : sem
↪  a :=
if p then s else mzero

definition check_unsigned (bits : ℕ) (x : nat) : sem nat :=
sem.guard (is_bounded_nat bits x) (return x)

definition checked.add (bits : ℕ) (x y : nat) : sem nat :=
check_unsigned bits (x+y)

...
```

We can avoid the check in operations that cannot overflow, such as unsigned division. We still have to check for division by zero, of course.

[8]When overflowing is indeed intended, the programmer may use special methods such as `u8::wrapping_add`

```
definition checked.div (bits : ℕ) (x y : nat) : sem nat :=
sem.guard (y ≠ 0) (return (div x y))
```

The signed implementations are equivalent, except that we do have to check for overflow during signed division by -1.

4.8.2 Bitwise Operators

We implement all bitwise operations on integral types by converting them to and from the `bitvec` type, which we adapted from the Lean standard library and expanded significantly.

```
abbreviation binary_bitwise_op (bits : ℕ) (op : bitvec bits → bitvec bits
↪  → bitvec bits)
  (a b : nat) : nat :=
bitvec.to ℕ (op (bitvec.of bits a) (bitvec.of bits b))

definition bitor bits := binary_bitwise_op bits bitvec.or
...
```

Some care must be taken when implementing bitwise shift: Shifting by the bitsize of a type or more bits will panic in Rust.

```
definition checked.shl [reducible] (bits : ℕ) (x : nat) (y : u32) : sem
↪  nat :=
sem.guard (y < bits)
  (return (bitvec.to ℕ (bitvec.shl (bitvec.of bits x) y)))
```

4.8.3 Index Expressions

While indexing is desugared to a call to the `Index` trait for most types, it is a primitive operation on arrays. Out-of-bounds accesses will panic in Rust. By identifying arrays with Lean lists, we can use the existing `list.nth` function and *lifting* its result into the semantics monad.

```
definition sem.lift_opt {A : Type₁} : option A → sem A
| none   := mzero
| some a := return
```

4.8.4 Lambda Expressions

Each lambda expression in Rust has a unique type that represents its *closure*, the set of variables captured from the outer scope. As necessitated by its ownership and mutability tracking, Rust files each closure type into one of three traits that together form a hierarchy:

```
pub trait FnOnce<Args> {
  type Output;
  fn call_once(self, args: Args) -> Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
  fn call_mut(&mut self, args: Args) -> Output;
}

pub trait Fn<Args> : FnMut<Args> {
  fn call(&self, args: Args) -> Output;
}
```

Rust will automatically infer the most general trait based on the lambda expression's requirements: If it has to move ownership of a captured variable, it can only implement `FnOnce`; if it needs a mutable reference to a variable, it can only implement `FnMut`; otherwise, it will implement `Fn`.

Because we lose the restrictions of linear typing during our translation, we can simplify the hierarchy: `FnOnce` can be implemented using `FnMut`, if implemented, which in turn can be implemented using `Fn` (because the closure must be immutable in that case).

```
structure FnOnce [class] (Self Args Output : Type₁) :=
(call_once : Self → Args → sem Output)

structure FnMut [class] (Self Args Output : Type₁) :=
(call_mut : Self → Args → sem (Output × Self))

definition FnMut_to_FnOnce [instance] (Self Args Output : Type₁)
  [FnMut Self Args Output] : FnOnce Self Args Output :=
⦃FnOnce, call_once := λ self args, do x ← FnMut.call_mut _ self args;
   return x.1⦄

structure Fn [class] (Self : Type₁) (Args : Type₁) (Output : Type₁) :=
(call : Self → Args → sem Output)

definition Fn_to_FnMut [instance] (Self Args Output : Type₁) [Fn Self Args
↪  Output]
  : FnMut Self Args Output :=
⦃FnMut, call_mut := λ self args, do x ← Fn.call _ self args;
  return (x, self)⦄
```

Translating a lambda expression means declaring a closure type according to the captured environment and creating a trait implementation according to the closure kind as reported by the compiler. Calling a lambda expression, on the other hand, is no different from other trait method calls and does not need any special casing.

```rust
fn f(x: i32) -> i32 {
  let l = |y| x + y;
  l(x)
}
```

```lean
structure f.closure_7 := (val : i32)

definition f.closure_7.fn (c : f.closure_7) (y
↪ : i32) : sem i32 :=
let t4 := f.closure_7.val c in
checked.sadd i32.bits t4 y

definition f.closure_7.inst [instance] : Fn
↪ f.closure_7 i32 i32 :=
⦃Fn, call := f.closure_7.fn⦄

definition f (x : i32) : sem i32 :=
let l := f.closure_7.mk x in
sem.incr 1 (Fn.call _ l x)
```

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

As a first test of the translation tool, we set out to verify the correctness of the binary search implementation in the Rust standard library, an algorithm of medium complexity.

5.1 The Rust Implementation

Before we can even tackle the algorithmic complexity, we have to cope with the design complexity of a real-world library. The public implementation of the `binary_search` method implemented on any slice type can be found in the `collections` crate.

```
use core::slice as core_slice;

impl<T> [T] {
  ...

  /// Binary search a sorted slice for a given element.
  ///
  /// If the value is found then `Ok` is returned, containing the
  /// index of the matching element; if the value is not found then
  /// `Err` is returned, containing the index where a matching
  /// element could be inserted while maintaining sorted order.
  ///
  /// ...
  pub fn binary_search(&self, x: &T) -> Result<usize, usize>
    where T: Ord {
    core_slice::SliceExt::binary_search(self, x)
  }
}
```

As we can see from the its documentation and signature, the method is very general: It works on all slices whose element type implements the `Ord` trait, and it returns information in both the success and the failure case. The implementation, however, turns out to be merely a redirection to a trait method in the base crate `core`. This trait has a single implementation, for the slice type.

```
pub trait SliceExt {
  type Item;

  fn binary_search(&self, x: &Item) -> Result<usize, usize>
    where Item: Ord;
  fn len(&self) -> usize;
  fn is_empty(&self) -> bool { self.len() == 0 }
  ...
}
```

```
fn binary_search_by<'a, F>(&'a self, mut f: F) -> Result<usize, usize>
    where F: FnMut(&'a T) -> Ordering
{
    let mut base = 0usize;
    let mut s = self;

    loop {
        let (head, tail) = s.split_at(s.len() >> 1);
        if tail.is_empty() {
            return Err(base)
        }
        match f(&tail[0]) {
            Less => {
                base += head.len() + 1;
                s = &tail[1..];
            }
            Greater => s = head,
            Equal => return Ok(base + head.len()),
        }
    }
}
```

Listing 3: Implementation of the `binary_search_by` method. A subslice `s` of `self` is iteratively bisected until it is empty or the element has been found. The `tail[1..]` *slicing syntax* is syntax sugar for `tail.index(RangeFrom{start: 1})`.

```
impl<T> SliceExt for [T] {
  type Item = T;

  fn binary_search(&self, x: &T) -> Result<usize, usize> where T: Ord {
    self.binary_search_by(|p| p.cmp(x))
  }
  fn len(&self) -> usize { ... }
  ...

}
```

This indirection seems pointless at first, but follows from a technical restriction: There may be at most one `impl` block for a primitive type like `[T]`. Because the `core` crate does not depend on the existence of a heap allocator, but some methods on `[T]` like its merge sort implementation do need dynamic allocation, the `impl` block is only declared in the later `collections` crate. Since `binary_search` does not need an allocator, it should still reside in `core`, and instead is associated to the slice type via the helper trait.

This final version of `binary_search`, which we represent as `core::<[T] as SliceExt>::binary_search`, is implemented by way of a more general method `binary_search_by` that takes a comparison function instead of being constrained to `Ord` (Listing 3). This method, finally, turns out to be much more abstract than one might expect: Instead of the standard binary search implementation that iteratively reduces the search range via two indices, the range is represented as a subslice and manipulated via high-level slice methods such as `split_at`. The reasoning behind this is a great show case for Rust's zero-cost (or even negative-cost, in this case) abstractions philosophy – the abstract implementation actually surpasses a direct implementation in terms of efficiency because it helps the compiler to eliminate all bounds checks in it. It also elegantly avoids the common pitfall[9] of a potential integer overflow in less abstract code like `mid = (low + high) / 2`.

[9]https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

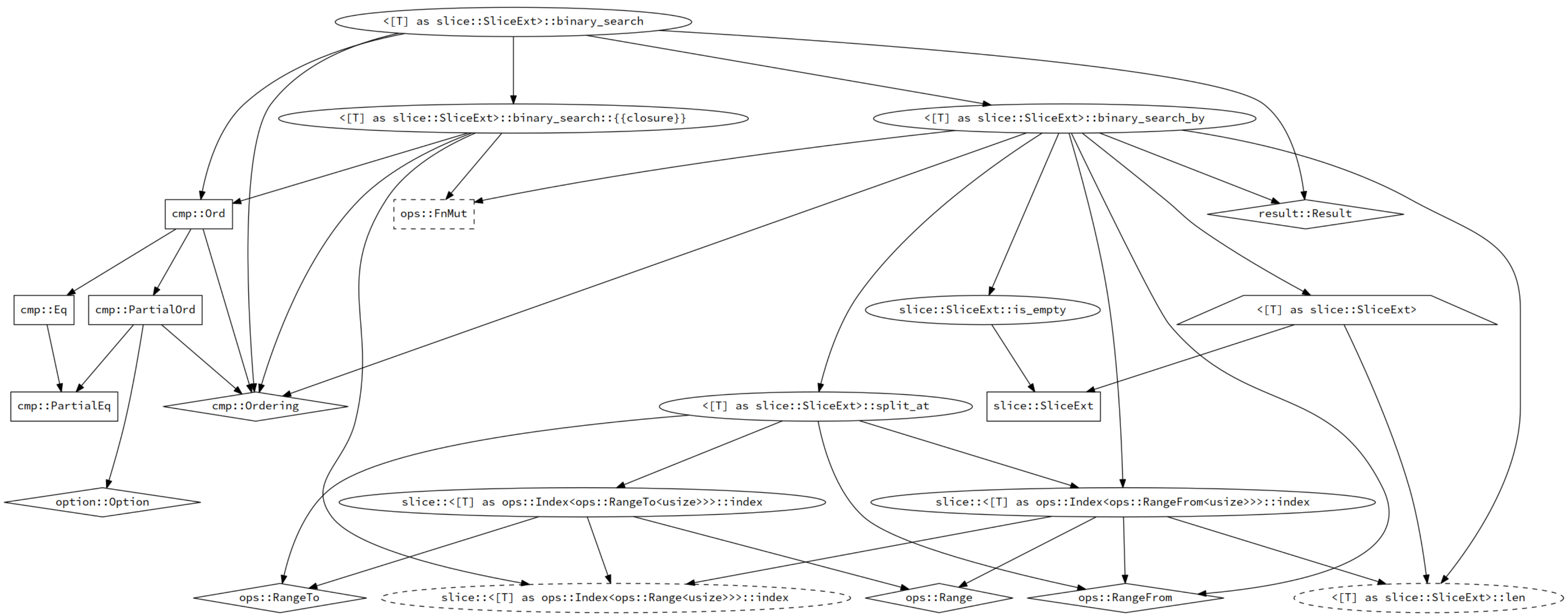


Figure 4: A complete graph of the translation dependencies of `binary_search` in the `core` crate, distinguishing between functions, types, traits, and trait implementations. Axiomatized items that use unsafe code in the original implementation are marked by dashed borders. Because we eagerly resolve trait method calls where possible, such as to the `index` method of `Index<RangeFrom<usize>>` for `[T]`, we can avoid some dependencies like the full `Index` implementation for `[T]`, and even the trait itself.

For our purposes, the abstract implementation primarily means a fair number of additional dependencies we have to support and inspect (Figure 4). All in all, `binary_search` turned out to be an ideal first test not only because of its algorithmic complexity, but also because of its use of numerous Rust language features including enums, structs, traits with associated types and default methods, higher-order functions, and loops.

5.2 Prelude: Coping with Unsafe Dependencies

When trying to translate the `binary_search` method including its dependencies, we will not get back a working definition at first. Our tool refuses to translate some dependencies because they use unsafe code, as marked in Figure 4. We will have to translate these functions manually, basically adding the correctness of their translation as axioms to the project.

Apart from our custom translation of `FnMut` we discussed in Subsection 4.8.4, both axiomatized functions operate on slices and are straightforward to implement using our identification of slices with Lean lists.

```
-- Returns the number of elements in the slice.
definition «[T] as core.slice.SliceExt».len {T : Type₁} (self : slice T) :
↪  sem nat :=
return (list.length self)

-- Implements slicing with syntax `&self[begin .. end]`.
-- Returns a slice of self for the index range [`begin`..`end`).
-- This operation is `O(1)`.
-- Requires that `begin <= end` and `end <= self.len()`,
-- otherwise slicing will panic.
definition «[T] as core.ops.Index<core.ops.Range<usize>>».index {T : Type₁}
↪  (self : slice T) (index : Range usize) : sem (slice T) :=
sem.guard (Range.start index ≤ Range.«end» index ∧ Range.«end» index ≤
↪  list.length self)
  (return (list.firstn (Range.«end» index - Range.start index) (list.dropn
↪  (Range.start index) self)))
```

The latter method presents a small technical hurdle: It is dependent on other translation products, specifically the `Range` structure. Instead of having to axiomatize that perfectly translatable item and adding both definitions manually to the `pre.lean` file, we instruct the translator in the `config.toml` file to inject our Lean definition of `index` as the translation of the Rust definition on-the-fly.

```
[replace]
"«[T] as core.ops.Index<core.ops.Range<usize>>».index" = "..."
```

5.3 Formal Specification

Going back to the original definition of `[T]::binary_search`, we translate the documented behavior into a Lean predicate.

```
parameters {T : Type₁} [Ord T]
parameter self : slice T
parameter needle : T -- a more descriptive name for the parameter `x`

inductive binary_search_res : Result usize usize → Prop :=
| found     : Πi, list.nth self i = some needle → binary_search_res
↪  (Result.Ok i)
| not_found : Πi, needle ∉ self → sorted (list.insert_at self i needle) →
  binary_search_res (Result.Err i)
```

It is specifications like these where the power of shallow embeddings really shines: We can freely mix and match Rust types and standard Lean functions and constructs. In fact, we will have to do some more mixing of these two worlds to make the definition valid: While we have copied the assumption `T : Ord` from the `binary_search` method, the `sorted` predicate expects T to implement Lean's own ordering typeclass. We therefore introduce a new typeclass `Ord'` that merges both typeclasses – or rather, in the Lean case, the subclass of decidable, linear orders.

```
definition ordering {T : Type} [decidable_linear_order T] (x y : T) :
↪  cmp.Ordering :=
if x < y then Ordering.Less
else if x = y then Ordering.Equal
else Ordering.Greater

structure Ord' [class] (T : Type₁) extends Ord T, decidable_linear_order T
↪  :=
(cmp_eq : ∀ x y : T, Σ k, cmp x y = some (ordering x y, k))
```

After changing the **`parameter`** definition to `Ord' T`, the specification type-checks. We need two more (sensible) hypotheses before we can prove that `binary_search` upholds the specification.

```
hypothesis Hsorted : sorted self
hypothesis His_slice : is_slice self

...

theorem binary_search.spec : sem.terminates_with
  binary_search_res
  (binary_search self needle) := ...
```

5.4 Proof

The full correctness proof is about 170 lines in Lean's tactic mode. We will not discuss the individual steps or the Lean tactic syntax here, but focus on the main proof steps.

After unfolding the `binary_search` and `binary_search_by` definitions and some simplifications, we quickly reduce the proof obligation down to the central loop.

```
⊢ sem.terminates_with binary_search_res
    (loop loop_4 (closure_5594.mk needle, 0, self))
```

Here `loop_4` is the loop body extracted from `binary_search_by`, which is passed to the loop combinator `loop` together with the initial loop state. The loop state is the triple `(f, base, s)` of local variables mutated in the loop, initialized to the closure from `binary_search` (capturing `needle`), `0`, and `self`, respectively. As described in Subsection 4.7.1, we can reduce the goal to one basing the loop on a specific relation by use of the lemma `loop.fix_eq_loop`.

```
abbreviation f₀ := closure_5594.mk needle
abbreviation loop_4.state := closure_5594 T × usize × slice T
definition R := measure (λ st : loop_4.state, length st.2)
...

⊢ sem.terminates_with binary_search_res
  (loop.fix loop_4 R (f₀, 0, self))
```

`measure` lets us create a well-founded relation on the loop state triple by comparing the length of `s`. We will not be able to show the new goal directly via well-founded induction over `R`, instead we first have to generalize it. For that we first declare the loop invariants (which we obtained by the non-sophisticated method of repeated try-and-error).

```
variables (base : usize) (s : slice T)

structure loop_4_invar :=
(s_in_self  : s ⊑ₚ (dropn base self))
(insert_pos : sorted.insert_pos self needle ∈ '[base, base + length s])
(needle_mem : needle ∈ self → needle ∈ s)
```

These say that

1. `s` is a contiguous subsequence of the original slice `self` starting at `base`; here $\sqsubseteq_p$ is a notation for the (non-strict) list prefix order that will come in handy at multiple points in the proof.

2. inserting `needle` at the first position in `self` that will keep it sorted will insert it inside or adjacent to `s`.

3. if `needle` is at all in the original slice, it will also be in `s`. If this is the case, this invariant will imply the previous one, but in general they are independent.

Because the invariants trivially hold for the initial state, we can generalize the goal.

```
⊢ loop_4_invar base s → sem.terminates_with binary_search_res
    (loop.fix loop_4 R (f₀, base, s))
```

There is no need to generalize `f₀` because we know it is a non-modifying closure and thus the variable `f` will always contain that value.

After applying well-founded recursion, we unroll one iteration of `loop.fix` via the lemma `loop.fix_eq` from Subsection 4.7.1 and apply the induction hypothesis on the loop remainder to reduce the goal to that single iteration.

```
inductive loop_4_step : loop_4.state → Prop :=
mk : Π base' s', loop_4_invar base' s' → length s' ≤ length s / 2 → length
↪  s ≠ 0 →
  loop_4_step (f₀, base', s')

abbreviation loop_4_res := sum.rec (loop_4_step s) binary_search_res

⊢ loop_4_invar base s → sem.terminates_with
    loop_4_res
    (loop_4 (f₀, base, s))
```

If the iteration breaks the loop (returns some `sum.inr`), we need the result to fulfill the top-level specification `binary_search_res`. Otherwise, if the loop produces some new loop state `(f₀, base', s')`, the loop invariants should be upheld together with a loop *variant* saying that the length of `s` has at least halved. Together with the information that `length s ≠ 0`, this implies `length s' < length s` and ensures we can apply the induction hypothesis. We will need the former two stronger statements for proving the function's logarithmic complexity below.

The remainder of the proof, while tedious, uses mostly basic reasoning. We split the goal according to the **`if`** and **`match`** branches in the original code and, depending on the return value in each case, show that `loop_4_invar` or `binary_search_res` is upheld. We prove that neither of the two additions in the code overflows by showing that they are bounded by `list.length self`, which by the assumption `is_slice self` fits into the **`usize`** type.

6 Transformation of Mutable References

As the previous section showed, the basic transformation already allows us to reason about mutability in form of local variables, including inside loops. The next step is to support non-local or indirect mutability in form of mutable references. We will develop a restricted but extendable transformation of mutable references in this section and put it to use in the next section.

6.1 Lenses as Functional References

In order to correctly translate mutable references, we will take a more careful look at their structure in the MIR (Subsection 4.1). Mutable references are created by the `&mut x` syntax, which in MIR operates on *lvalues*.

```
pub enum Rvalue<'tcx> {
  /// &x or &mut x
  Ref(&'tcx Region, BorrowKind, Lvalue<'tcx>),
  ...
}
```

An lvalue in Rust is either a local or static (global) variable, or inductively some projection of another lvalue.

```
pub enum Lvalue<'tcx> {
    Local(Local),
    Static(DefId),
    /// projection out of an lvalue (access a field, deref a pointer, etc)
    Projection(Box<LvalueProjection<'tcx>>),
}
```

Because mutable static variables are not allowed in safe Rust, we may assume that every lvalue is rooted in a local variable. We can describe a mutable reference as *focusing* on some part of a local variable, which in functional programming can represented by *lenses* [9] (also known as *functional references*). For our purposes, a very simple presentation of lenses that allows us to get and set the focused part is sufficient. We also specialize it to return our semantics monad.

```
structure lens (Outer Inner : Type₁) :=
(get : Outer → sem Inner)
(set : Outer → Inner → sem Outer)
```

Our lens type describes how some type `Inner` can be extracted from and replaced inside another type `Outer`. For the correct combinations of those two types, we can give some general instances such as identity and composition.

```
definition lens.id {Inner : Type₁} : lens Inner Inner :=
⦃lens, get := return, set := λ o, return⦄

definition lens.comp {A B C : Type₁} (l₂ : lens B C) (l₁ : lens A B) : lens
↪  A C :=
⦃lens, get := λ o,
  do o' ← lens.get l₁ o;
  lens.get l₂ o',
 set := λ o i,
  do o' ← lens.get l₁ o;
  do o' ← lens.set l₂ o' i;
  lens.set l₁ o o'⦄

infixr ` ∘ₗ `:60 := lens.comp
```

With this, we can translate the `&mut x` operation: We generate a lens per projection, then compose them together to obtain a value of type `lens A B` where `B` is the type of `x`, and `A` the type of the root variable of `x`. For the projection of indexing into an array or slice we can give a generic definition, but for other projections such as struct fields we will have to generate them at translation time.

```
definition lens.index (Inner : Type₁) (index : ℕ) : lens (slice Inner)
↪  Inner :=
⦃lens,
  get := λ o, sem.lift_opt (list.nth self o),
  set := λ o i, sem.lift_opt (list.update o index i)⦄
```

There is one projection we have to special case: dereferencing an lvalue as in `*x`. If `x` is an immutable reference, this is just the identity lens because `&T` and `T` are translated to the same type. If it is a mutable reference, we compose with its lens to obtain a lens on the ultimate root variable. This combination of referencing and dereferencing is also known as “reborrowing”.

```
let x: &mut [i32] = &mut
↪  a;
let y = &mut (*x)[1];
```

```
let x := lens.id in
let y := lens.index _ 1 ∘ₗ x in
...
```

There is a final technicality involved with creating mutable references. Because in Rust a reference is represented merely by an address, index projections are checked to be in bounds when creating the reference, whereas `lens.index` will return `mzero` only when its getter or setter is used. Therefore, we “probe” lenses eagerly after creation by invoking their getter in order to make sure we exhibit the same termination behavior as the original code.

6.2 Pointer Bookkeeping

In order to actually invoke `lens.get` or `lens.set`, we also need to pass it the "outer" object, i.e. the root variable of the original borrow. This is not a kind of information we can dynamically save alongside the lens in the mutable reference, but we instead have to statically determine at translation time. For now, we represent this information as a mapping from variable names to variable names.

```
let x: &mut [i32] = &mut a; // {'x' ~> 'a'}
// moving &mut
let x2 = x;                 // {'x2' ~> 'a'}
// reborrowing &mut
let y = &mut (*x2)[1];      // {'x2' ~> 'a', 'y' ~> 'a'}
```

While this simple mapping has proved sufficient so far, it does impose the following limitations:

- Mutable references can only be stored directly in variables, not nested in some structure. This also means that we do not have to worry about how to represent mutable references in data types, yet.
- Whereas a completely static mapping works for linear code, it cannot work for variables that are part of a loop state in general. We could lift this restriction for the most common special case where the loop changes the lens, but not the root variable of a reference.

6.3 Passing Mutable References

In Subsection 3.1, we introduced references as a more ergonomic (and efficient) way of passing ownership of a value to some function and getting back the old or (in the case of mutable references) new value from the function. While we do not have to worry about ownership in Lean, we can still invert this pattern for passing mutable references in Lean. For each mutable reference argument, we read the current value through the lens, pass it to the function, get back the new value as part of the return value, and write it back through the lens. Inside the called function, we immediately re-wrap the value in the identity lens.

```
fn f(x: &mut T) -> R {...}
...

let x: &mut T = &mut
↪  ...a...;
let y = f(x);
```

```
definition f : (xₐ : T) : sem (R × T) :=
let x := lens.id in -- {'x' ~> 'xₐ'}
...

let x = ... in
do tmp ← lens.get x a;
do ret ← f tmp;
match ret with
| (y, tmp) :=
  do a ← lens.set x a tmp;
  ...
end
```

There is a small caveat with this approach: It does not work if a parameter's type is declared to be a type parameter, but then instantiated to a mutable reference.

6.4 Returning Mutable References

While passing mutable references to functions has a rather simple desugaring, returning them is a very different beast altogether: The caller has no idea where the reference is pointing to. For now, we restrict ourselves to the special case of returning mutable references that point into the first parameter, which in particular covers all methods that return references pointing into their `&mut self` parameter. We statically check this property when translating the callee, and then use that knowledge in the caller to compose the returned lens with the lens of the first argument. Note that we still have to return the new pointee for the first argument, as by the previous subsection.

```
fn f(x: &mut T) -> &mut R
↪  {...}
...

let x: &mut T = &mut
↪  ...a...;
let y = f(x);
```

```
definition f : (xₐ : T) : sem (lens T R × T)
↪  := ...

let x = ... in
do tmp ← lens.get x a;
do ret ← f tmp;
match ret with
| (y, tmp) :=
  let y := y ∘ₗ x in
  do a ← lens.set x a tmp;
  ...
end
```

7 Case Study: Partial Verification of `FixedBitSet`

Whereas our first case study focused on algorithmic verification, for our second study we chose the `FixedBitSet` data structure from the `fixedbitset` crate[10]. It can be thought of as a more efficient version of `Vec<bool>` that stores elements packed at the bit level. While it is not a complex data structure, verifying it does require reasoning about the following important parts:

- The ubiquitous `Vec` type from the standard library, which `FixedBitSet` uses internally, including mutable references into it
- Data structure invariants
- Bitwise operations

7.1 The Rust Implementation

`FixedBitSet` uses an internal `Vec` to store up to 32 bits per element.

```
type Block = u32;

pub struct FixedBitSet {
  data: Vec<Block>,
  /// length in bits
  length: usize,
}
```

We will focus on three basic operations: creating (`with_capacity`), manipulating (`insert`), and querying (`contains`) a `FixedBitSet`. The Rust implementations are shown in Listing 4.

7.2 Prelude: Axiomatizing `collections::vec::Vec`

`Vec` is the standard type for dynamically-sized arrays in Rust. It is implemented on top of an unsafe abstraction called `RawVec` that handles allocating, resizing, and deallocating the array memory. `Vec` provides a safe interface on top of that type by additionally keeping track of ownership of individual items via a `len` field. Elements after the first `len` items are not logically part of the `Vec` and must be viewed as unitialized storage.

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

[10] https://docs.rs/fixedbitset/

```
const BITS: usize = 32;
fn div_rem(x: usize, d: usize) -> (usize, usize) {
  (x / d, x % d)
}

impl FixedBitSet {
  pub fn with_capacity(bits: usize) -> Self {
    let (mut blocks, rem) = div_rem(bits, BITS);
    blocks += (rem > 0) as usize;
    FixedBitSet {
      data: vec![0; blocks],
      length: bits,
    }
  }

  pub fn insert(&mut self, bit: usize) {
    assert!(bit < self.length);
    let (block, i) = div_rem(bit, BITS);
    unsafe {
      *self.data.get_unchecked_mut(block) |= 1 << i;
    }
  }

  pub fn contains(&self, bit: usize) -> bool {
    let (block, i) = div_rem(bit, BITS);
    match self.data.get(block) {
      None => false,
      Some(b) => (b & (1 << i)) != 0,
    }
  }
  ...
}
```

Listing 4: The Rust implementations of the three methods

Because `Vec` provides a safe interface, but is itself implemented using (predominantly) unsafe code, we both can and have to axiomatize it. When axiomatizing data structures, we are free to choose any abstraction as long as the operations on it preserve their semantics. Just as with arrays and slices, Lean's basic `list` type is a natural representation for `Vec`.

```
structure Vec (T : Type₁) :=
(buf : list T)
```

We do lose information about the `RawVec`'s length (also called the `Vec`'s *capacity*) here, but this information is not exposed by the `Vec` operations `FixedBitSet` depends on.

```
namespace «Vec<T>»
  parameter {T : Type₁}

  definition new : sem (Vec T) :=
  return (Vec.mk [])

  -- note: only a runtime upper bound
  definition push (self : Vec T) (value : T) : sem (unit × Vec T) :=
  sem.incr (list.length (Vec.buf self)) (return (unit.star, Vec.mk
↪  (Vec.buf self ++ [value])))

  -- note: `pop` never resizes the `Vec`, so it is always constant-time
  definition pop (self : Vec T) : sem (Vec T × Option T) :=
  match reverse (Vec.buf self) with
  | x :: xs := return (Vec.mk (reverse xs), Option.Some x)
  | []      := return (self, Option.None)
  end

  definition clear (self : Vec T) : sem (Vec T) :=
  sem.incr (list.length (Vec.buf self)) new

  definition len (self : Vec T) : sem usize :=
  return (list.length (Vec.buf self))
end «Vec<T>»
```

Listing 5: Axiomatizations of relevant `Vec` methods

All these operations are implemented using unsafe code, so we will have to axiomatize all of them too. Listing 5 lists the Lean implementations of the needed `Vec` methods, none of which should be surprising.

`Vec` also implements the `Deref` trait, which makes values of type `&Vec<T>` automatically coerce to `&[T]` and is implicitly being used in `FixedBitSet::contains`. This is easy enough to implement using our abstraction.

```
definition «collections.vec.Vec<T> as core.ops.Deref».deref {T : Type₁}
↪  (self : Vec T) :
  sem (slice T) :=
return (Vec.buf self)
```

There is also a corresponding `DerefMut` trait that makes `&mut Vec<T>` coerce to `&mut [T]`. The implementation is slightly more interesting because it has to return a lens focusing on `Vec.buf`.

```
definition «collections.vec.Vec<T> as core.ops.DerefMut».deref_mut {T :
↪  Type₁} (self : Vec T) :
  sem (lens (Vec T) (slice T) × Vec T) :=
return (⦃lens, get := return ∘ Vec.buf, set := λ old, return ∘ Vec.mk⦄,
        self)
```

This trait implementation is being being used by `FixedBitSet::insert` to access `[T]::get_unchecked_mut`, which in turn returns a mutable reference to a slice element.

```
definition «[T]».get_unchecked_mut {T : Type₁} (self : slice T) (index :
↪  usize) :
  sem (lens (slice T) T × slice T) :=
sem.guard (index < length self) (return (lens.index _ index, self))
```

This method is interesting in that it actually is unsafe to call in Rust – instead of an explicit panic, an out-of-bounds access will silently invoke undefined behavior.

```
unsafe fn get_unchecked_mut(&mut self, index: usize) -> &mut T {
    &mut *self.as_mut_ptr().offset(index as isize)
}
```

There is also a safe, panicking variant called `[T]::get_mut`, which we first mentioned in Section 2 as not being expressible in other verifiable languages. Because our semantics monad does not differentiate between undefined behavior and panics, both functions become semantically equivalent in our transformation and we can translate calls to both of them, including the small bit of unsafe code in `FixedBitSet::insert`.

7.3 Formal Specification

There is no useful abstract specification we could give `contains` without essentially restating its implementation. Instead, we use it to *build* an abstraction: We translate `FixedBitSet` to a Lean set of indices.

```
abbreviation sem.returns {A : Type₁} (x : A) := sem.terminates_with (λ a, a
↪  = x)

open FixedBitSet

definition to_set (s : FixedBitSet) : set usize :=
{bit | bit < length s ∧ sem.returns bool.tt (contains s bit)}
```

The additional constraint `bit < length s` may seem superfluous considering that `contains` makes sure to always return **`false`** for indices after the last **`u32`** block. However, indices between the length and the capacity may not necessarily be **`false`**, as noted in the docstring for a different method:

```
/// View the bitset as a mutable slice of `u32` blocks. Writing past the
bitlength in the last block
/// will cause `contains` to return potentially incorrect results for bits
↪  past the bitlength.
pub fn as_mut_slice(&mut self) -> &mut [u32]
```

With `to_set`, we can give `insert` a natural specification using standard Lean `set` operations.

```
lemma insert.spec (s : FixedBitSet) (bit : usize) : bit < length s →
  sem.terminates_with
    (λ ret,
      let s' := ret.2 in
      to_set s' = to_set s ∪ '{bit})
    (insert s bit)
```

To prove this lemma, we will also need a data type invariant on `FixedBitSet` relating its two fields: The `Vec` should always have the minimum length, that is, the number of bits divided by 32, then rounded up. As with traits, we specify the invariant as a type class.

```
structure FixedBitSet' [class] (self : FixedBitSet) : Prop :=
(length_eq : nat.div_ceil (length self) 32 = list.length (Vec.buf (data
↪  self)))
```

This invariant should be fulfilled by the only constructor, `with_capacity`.

```
lemma with_capacity_inv (bits : usize) [is_usize bits] :
  sem.terminates_with FixedBitSet' (with_capacity bits)
```

After adding the hypothesis `[FixedBitSet' s]` to `insert.spec`, the lemma becomes provable. We also show that the invariant is upheld, i.e. that `FixedBitSet' s'` holds.

7.4 Proof

We will focus on the correctness proof of `insert`. With 77 lines, it is quite shorter (and simpler) than the binary search proof, so we will show some more details, including some reasoning about bitwise operations.

We again start by unfolding definitions and simplifying the resulting goal. We also eliminate some bounds checks, introducing `bit_block` for the `u32` block `bit` is part of, and `l'` and `s'` for the updated list and `FixedBitSet`, respectively.

```
...
bit_block : ℕ,
bit_block_eq : list.nth (Vec.buf (FixedBitSet.data s)) (bit / 32) = some
↪  bit_block,
l' : list ℕ,
l'_eq : list.update (Vec.buf (FixedBitSet.data s)) (bit / 32) (bit_block
↪  ||[32] 2 ^ (bit % 32)) = some l',
s' : FixedBitSet,
s'_eq : s' = FixedBitSet.mk (Vec.mk l') (FixedBitSet.length s)
⊢ FixedBitSet' s' ∧ to_set s' = to_set s ∪ '{bit}
```

Here the notation `||[32]` is an abbreviation for `bitor 32`. We show the data type invariant by a helper lemma saying that `list.length` is invariant under `list.update`. After unfolding `to_set` and some more simplifications, we are left with a goal that asserts that some index `bit'` is in the new set iff it is in the old set or is equal to `bit`.

```
...
bit' : ℕ
⊢ bit' < FixedBitSet.length s ∧ sem.returns bool.tt (FixedBitSet.contains
↪  s' bit') ↔
  bit' < FixedBitSet.length s ∧ sem.returns bool.tt (FixedBitSet.contains
↪  s bit') ∨ bit' = bit
```

If `bit'` is not a valid index (`bit' ≥ FixedBitSet.length s`), the goal reduces to `bit' ≠ bit`, which holds because `bit` is assumed to be valid. If, on the other hand, `bit'` is valid, we will have to reason about the two `contains` calls. After unfolding them and some more simplifications, we are left with a bit-level goal talking about the `u32` block for `bit'` in the old set (`bit'_block`) and in the new set (`bit'_block'`), respectively.

```
...
bit'_block bit'_block' : ℕ,
bit'_block_eq : list.nth (Vec.buf (FixedBitSet.data s)) (bit' / 32) = some
↪  bit'_block,
bit'_block'_eq : (if bit / 32 = bit' / 32 then some (bit_block ||[32] 2 ^
↪  (bit % 32)) else some bit'_block) = some bit'_block'
⊢ bit'_block' &&[32] 2 ^ (bit' % 32) ≠ 0 ↔
  bit'_block &&[32] 2 ^ (bit' % 32) ≠ 0 ∨ bit' = bit
```

We proceed by splitting the goal according to the conditional in `bit'_block'_eq`. In the case `bit / 32 ≠ bit' / 32`, we obtain `bit'_block' = bit'_block` and `bit' ≠ bit`, closing the goal. In less formal words, `bit'` turned out to be in a block entirely unaffected by the whole insertion.

In the other case, we get `bit'_block = bit_block` and the goal reduces to a proposition about two bits in the same block.

```
⊢ (bit_block ||[32] 2 ^ (bit % 32)) &&[32] 2 ^ (bit' % 32) ≠ 0 ↔
  bit_block &&[32] 2 ^ (bit' % 32) ≠ 0 ∨ bit' = bit
```

Assuming `bit' = bit`, we see that both sides of the equivalence become universally true. Otherwise, if `bit' ≠ bit`, but `bit / 32 = bit' / 32` by the previous assumption, we obtain `bit' % 32 ≠ bit % 32`. A helper lemma proves that this cancels out the bitwise or and thus reduces both sides to the same term, concluding the proof.

8 Asymptotic Complexity Analysis

Monads are known for their versatility in representing various semantics, including side effects. So far, we have made use of our semantics monad for representing partiality, i.e. nontermination and abnormal termination. We may in the future extend the monad to reason about more effects such as (unsafe) mutable global variables or I/O. In this section, we instead make use of the monad for verifying a property different from functional correctness: runtime complexity.

8.1 Classifying Asymptotic Complexity

Our formalization of multiparametric asymptotic function analysis is based on the technical report *Formal Verification of Asymptotic Complexity Bounds for OCaml Programs* [13]. The main insight of the report is that we can elegantly formalize the notation of "going to infinity" for an arbitrary number of parameters using the mathematical concept of *filters*, originally from topology.

We refer to the report for a detailed description on filters. Fortunately for us, the Lean library already includes a definition of filters on sets. We will only need the filter `at_infty` on natural numbers, the filter combinator `prod_filter`, which we developed, as well as the "eliminator" `eventually`.

```
definition at_infty : filter ℕ := ...
notation `[at ` `∞]` := at_infty

definition prod_filter {A B : Type} (Fa : filter A) (Fb : filter B) :
↪  filter (A × B) := ...
notation `[at ` `∞` ` × ` `∞]` := prod_filter [at ∞] [at ∞]

definition eventually {A : Type} (P : A → Prop) (F : filter A) : Prop :=
↪  ...
```

A proposition such as `eventually P [at ∞ × ∞]` then has the intuitive meaning of holding iff there exists a pair of natural numbers such that `P` holds for all (componentwise) larger pairs.

We can now formalize the notions of a function being non-strictly and strictly asymptotically bounded by another function, which directly lead to the usual notations as classes of functions.

```
namespace asymptotic
  parameters {A : Type} (F : filter A)
  variables  (e f g : A → ℕ)

  protected definition le : Prop := ∃c, eventually {a | f a ≤ c * g a} F
  protected definition lt : Prop := ∀c, eventually {a | c * f a ≤ g a} F
  protected definition equiv : Prop := le f g ∧ le g f
```

```
  definition ub  := {f | le f g}
  definition sub := {f | lt f g}
  definition lb  := {f | le g f}
  definition slb := {f | lt g f}

  notation `𝒪(` g `)` ` F := ub F g
  notation `(` g `)` ` F := sub F g
  notation `Ω(` g `)` ` F := lb F g
  notation `ω(` g `)` ` F := slb F g
  notation `Θ(` g `)` ` F := equiv F g
end asymptotic
```

With the notations in place, we can prove familiar lemmas about combining complexity bounds for arbitrary functions and filters, and about bounds for some specific functions and filters.

```
lemma ub_subset_ub (hf : f ∈ 𝒪(g) F) : 𝒪(f) F  𝒪(g) F := ...
lemma ub_add (h₁ : f₁ ∈ 𝒪(g) F) (h₂ : f₂ ∈ 𝒪(g) F) : f₁ + f₂ ∈ 𝒪(g) F :=
↪  ...
lemma ub_add_left (h : f ∈ 𝒪(g₂) F) : f ∈ 𝒪(g₁ + g₂) F := ...
lemma ub_add_const (h : f₁ ∈ 𝒪(g) F ∩ Ω(λ x, k) F) :
  f₁ + (λ x, k) ∈ 𝒪(g) F ∩ Ω(λ x, k) F := ..
lemma ub_const : (λa, k) ∈ 𝒪(1) F := ...
lemma ub_mul_prod_filter (h₁ : f₁ ∈ 𝒪(g₁) F₁) (h₂ : f₂ ∈ 𝒪(g₂) F₂) :
  (λp, f₁ p.1 * f₂ p.2) ∈ 𝒪(λp, g₁ p.1 * g₂ p.2) (prod_filter F₁ F₂) := ...


lemma log_unbounded {b : ℕ} (H : b > 1) : log b ∈ ω(1) [at ∞] := ...
lemma id_unbounded : id ∈ ω(1) [at ∞] := ...
```

8.2 Verifying the Asymptotic Complexity of `[T]::binary_search`

As described in Subsection 4.6, our semantics monad contains a step counter that is incremented on each function call and loop iteration. Because only a constant number of instructions can be executed between any such two events for a given program, the step count of an execution is asymptotically equivalent to the instruction count, which is usually assumed to be asymptotically equivalent to the running time.

We extend our existing correctness proof of `binary_search` by introducing a new predicate that tests both the return value and the step count.

```
inductive sem.terminates_with_in {a : Type₁} (H : a → Prop) (max_cost : ℕ)
↪  : sem a → Prop :=
mk : Π {x k}, H x → k ≤ max_cost → sem.terminates_with_in H max_cost (some
↪  (x, k))
```

Because we will only prove asymptotic upper bounds, we also use an upper bound in the definition in order to simplify reasoning about specific cost functions. If we wanted to use operators other than $\mathcal{O}$, we should turn the inequality into an equality.

This time we analyze the function bottom-up, starting with a single loop iteration, i.e. a call of `loop_4`. With all dependencies unfolded, we quickly obtain a constant bound on the step count for everything except the trait call to `Ord.cmp`, of whose complexity we have absolutely no information. In order to obtain the textbook bound of $\mathcal{O}(\log n)$ for binary search, we would have to assume that comparing two elements takes only constant time. That is certainly not true for all implementations of the trait and such a restriction would be a shame since we did a general correctness proof for any decidable linear order. Thus we instead introduce a more dynamic upper bound for the call: the maximum of all execution costs of such comparisons.

```
-- recall our extension of `Ord` from Section 5.3
structure Ord' [class] (T : Type₁) extends Ord T, decidable_linear_order T
↪  :=
(cmp_eq : ∀ x y : T, Σ k, cmp x y = some (ordering x y, k))

definition Ord'.cmp_max_cost {T : Type₁} [Ord' T] (y : T) (xs : list T) :=
-- extracts `k` from the above definition
Max x ∈ to_finset xs, sigma.pr1 (cmp_eq x y)
```

Now we can prove a specific upper bound of `Ord'.cmp_max_cost needle self + 15` for the loop body. Finally, we abstract from this explicit cost function to an asymptotic bound.

```
lemma loop_4.spec :
  ∃ c ∈ 𝒪(id) [at ∞],
  ∀ self needle s base, sorted le self → is_slice self → loop_4_invar self
↪  needle s base →
  sem.terminates_with_in
    (loop_4_res self needle s)
    (c (Ord'.cmp_max_cost needle self))
    (loop_4 (closure_5642.mk needle, base, s)) :=
exists.intro (λ n, n + 15) ...
```

This lemma says that the execution cost of `loop_4` is linearly bound by the maximum comparison cost. In general, we have to separate the *measure* function that reduces the input data to a natural number (here `cmp_max_cost`) and the (abstract or specific) cost function that describes the asymptotic behavior of the measure result, since we cannot define the latter on arbitrary domains. The composition of both then gives us the actual upper bound function.

We also have to make sure to introduce any parameters the measure depends on only after the existential quantifier. This makes the definitions slightly more verbose since we cannot use the convenient **section** mechanisms with them any more.

Going up, we expect the running time of the whole loop to be that of the body multiplied with `log₂ (length self)`. Formally, we again have to split the measure function `length` from the asymptotic cost function `log₂`.

```
lemma loop_loop_4.spec :
  ∃₀ f ∈ 𝒪(λp, log₂ p.1 * p.2) [at ∞ × ∞],
  ∀ self needle, is_slice self → sorted le self → sem.terminates_with_in
    (binary_search_res self needle)
    (f (length self, Ord'.cmp_max_cost needle self))
    (loop loop_4 (closure_5642.mk needle, 0, self)) := ...
```

As with the functional correctness proof, we can show this lemma by well-founded recursion. However, proving that a loop is asymptotically bounded by an iteration upper bound multiplied by an upper bound for the body should be a common occurence, so we have extracted the proof into a general theorem.

```
theorem loop.terminates_with_in_ub
  {In State Res : Type₁}
  (body : In → State → sem (State ⊕ Res))
  (pre : In → State → Prop)
  (p : In → State → State → Prop)
  (q : In → State → Res → Prop)
  (citer aiter : ℕ → ℕ)
  (miter : State → ℕ)
  (cbody abody : ℕ → ℕ)
  (mbody : In → State → ℕ)
  (citer_aiter : citer ∈ 𝒪(aiter) [at ∞] ∩ Ω(1) [at ∞])
  (cbody_abody : cbody ∈ 𝒪(abody) [at ∞] ∩ Ω(1) [at ∞])
  (pre_p : ∀ args s, pre args s → p args s s)
  (step : ∀ args init s, pre args init → p args init s →
    sem.terminates_with_in (λ x, match x with
      | inl s' := p args init s' ∧ citer (miter s') < citer (miter s)
      | inr r  := q args init r
    end) (cbody (mbody args init)) (body args s)) :
  ∃ f ∈ 𝒪(λ p, aiter p.1 * abody p.2) [at ∞ × ∞], ∀ args s, pre args s →
    sem.terminates_with_in (q args s) (f (miter s, mbody args s))
      (loop (body args) s) := ...
```

This may very well be the most complex theorem of our work, at least by signature. Going through the explicit parameters from top to bottom, we have the loop body, the precondition, the invariant (which may depend on both the initial and current state), the postcondition, the concrete and asymptotic bound and measure function of the iteration count and the same for the body. These are followed by assumptions that the asymptotic bounds are correct, that the precondition implies the invariant, and that a loop iteration either continues the loop with the invariant upheld and the concrete iteration count reduced or breaks the loop while satisfying the postcondition. In the end, the conclusion says that the loop, measured by the product of the measure functions, is asymptotically bounded by the product of the asymptotic bounds and terminates with the postcondition fulfilled, as long as the precondition is satisfied.

When using this theorem to prove the previous lemma, we can transfer the instantiations of and proofs about the precondition, invariant, and postcondition directly from the correctness proof, and show the asymptotic behavior of the body from the lemma `loop_4.spec`. We are left to prove that the iteration count is asymptotically bounded by `log₂`. Because this is the more interesting bound, we will show some more details of the proof.

We choose the concrete bound `λ n, log₂ (2 * n) + 1` for the iteration count and show that it is in `𝒪(log₂) [at ∞]`. Because the underlying relation `asymptotic.le` is transitive, we can make use of Lean's **`calc`** blocks for this.

```
local infix ``:25 := asymptotic.le [at ∞]
...

calc (λ a, log₂ (2 * a) + 1)
     (λ a, log₂ a + 2) : ub_of_eventually_le_at_infty 1 (
      take a, suppose a ≥ 1,
      calc log₂ (2 * a) + 1 = log₂ a + 1 + 1 : ...
                        ... ≤ log₂ a + 2     : ...)
...  log₂ : ub_add asymptotic.le.refl (
      calc (λ x, 2)  (λ x, 1) : ub_const
                 ...  log₂      : asymptotic.le_of_lt (log_unbounded ...))
```

Finally, we show that the conrete iteration count is strictly decreasing. This follows from the premises `length s' ≤ length s / 2` and `length s ≠ 0` of `loop_4_step` from the correctness proof, together with the fact that `log₂` is monotone.

```
calc log₂ (2 * length s') + 1
    ≤ log₂ (length s) + 1     : ...
... = log₂ (2 * length s)     : ...
... < log₂ (2 * length s) + 1 : le.refl
```

There is a single call going up from the loop to `binary_search`, leaving the final asymptotic complexity unchanged.

```
theorem binary_search.spec :
  ∃ f ∈ 𝒪(λ p, log₂ p.1 * p.2) [at ∞ × ∞],
  ∀(self : slice T) (needle : T), is_slice self → sorted le self →
↪  sem.terminates_with_in
```

```
(binary_search_res self needle)
(f (length self, Ord'.cmp_max_cost needle self))
(binary_search self needle) := ...
```

Thus, in the most general sense, the runtime of `binary_search` is asymptotically bounded by the logarithm of the input slice's length multiplied with the maximum cost of comparing the needle with any element in the slice.

9 Conclusion and Future Work

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] M. M. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *ACM SIGPLAN Notices*, volume 40, pages 1–13. ACM, 2005.

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

[6] 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.

[7] 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.

[8] 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.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[10] 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.

[11] 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.

[12] J. Guitton, J. Kanig, and Y. Moy. Why Hi-Lite Ada. 2011.

[13] A. Guéneau. Formal verification of asymptotic complexity bounds for OCaml programs. Technical report, Inria Paris-Rocquencourt, 2015.

[14] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, pages 230–244. Springer, 2000.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] A. Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[23] 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