TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Center for Advancing Electronics Dresden, Chair for Compiler Construction

Master Thesis

Verifying the Rust Runtime of Lingua Franca

Johannes Hayeß

Born on: 10th of November 1997 in Potsdam

to achieve the academic degree

Master of Science (M.Sc.)

First referee

Prof. Dr. Jeronimo Castrillon

Second referee

Dr. Steffen Märcker

Advisors

Christian Menard

Clément Fournier

Submitted on: 6th of March 2023

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur Compilerbau

Task Description for Master's Thesis

for:	**Johannes Hayeß**
Major:	Master Informatik
Matriculation Nr.:	4607666
Title:	**Verifying the Rust Runtime of Lingua Franca**

Lingua Franca (LF) is a polyglot coordination language implementing the programming model of "Reactors". This model borrows principles from well studied paradigms including actors, dataflow and discrete event systems, and enriches them with a timed semantics that makes the model particularly useful for designing applications in cyber-physical systems. Thereby, the reactor semantics guarantees a deterministic execution even in distributed systems.
The polyglot approach of Lingua Franca allows the reactor model to be used in multiple "back-ends" supporting different target languages. Each LF target provides a specific runtime that governs the execution of LF programs and enforces reactor semantics. In a recent effort, a Rust target was added to Lingua Franca. Rust is a particular attractive target language for LF due to its unique paradigm based on a strict typesystem with an ownership model. This enables the rust compiler to reason more precisely about execution steps, allowing, in principle, execution that is predictable.
Recent works (RustBelt, RustHornBelt, Creusot) have shown that Rust is also particularly useful for software verification. While verifying unsafe languages such as C/C++ is challenging due to the use of mutable pointers, the unique properties of Rust enable an encoding in first-order logic. In particular, Creusot has is capable of proofing complex properties in large Rust code bases. This has been recently demonstrated in CreuSAT, a verified and heavily optimized SAT solver.
The goal of this thesis is to leverage the capabilities of Creusot to proof the correctness of the LF Rust runtime. Ideally, the thesis would provide a complete proof, verifying that the Rust runtime indeed correctly implements reactor semantics. However, at the moment of this writing, it is unclear whether a complete proof is feasible within the scope of a Master's thesis. For this reason, this task only requires proofing selected properties of the runtime.
In particular, the student should understand and explain the semantics of Lingua Franca including the Rust runtime, as well as the capabilities and limitations of Creusot. Understanding these, the student should define one or multiple correctness criteria and develop a strategy for verifying those properties of the runtime. According to this strategy, the student should then use Creusot to annotate the runtime and aim at proofing the selected correctness criteria. Where possible and necessary, the software design of the runtime should be adjusted as is needed for the proof. Similarly, any bugs encountered during this work should be fixed if feasible. The student should further summarize the results and discuss to which extends the correctness criteria are met and how both the proof and the runtime can be further improved in future work.

Advisor:	Christian Menard, Clément Fournier
1. Examiner:	Prof. Dr. Jeronimo Castrillon
2. Examiner:	Dr. Steffen Märcker
Issued:	04. October, 2022

Prof. Dr. Christel Baier
Vorsitzender des Prüfungsausschusses

Prof. Dr. Jeronimo Castrillon
Verantwortlicher Hochschullehrer

Abstract

Lingua Franca is a coordination language for writing applications with deterministic concurrency in various programming languages, including Rust. For enforcement of semantics, Lingua Franca programs rely on a runtime. In the Rust case, the runtime is written in the same language, which provides memory safety and data-race freedom. To further capitalise on these benefits, this work defines correctness properties for the Lingua Franca Rust runtime and attempts to verify them with the deductive verification tool Creusot. Beyond that, the work surveys the contemporary academic literature on Rust software verification and its tooling. Ultimately, the verification work is only a limited success due to Creusot's lack of support for several Rust language features. The verification work required the extension of Creusot's contract coverage of Rust's standard library which was contributed back to the project. While full verification of the runtime failed, it was possible to verify the correctness of a custom map data type the runtime heavily relies on.

Declaration of authorship

I hereby declare that I have written this final thesis independently and have listed all used sources and aids. I am submitting this thesis for the first time as a piece of assessed academic work. I understand that attempted deceit will result in the failing grade “not sufficient” (5.0).

Dresden, 6th of March 2023
Johannes Hayeß

Johanne
s Hayess

Digital
unterschrieben von
Johannes Hayess
Datum: 2023.03.05
18:01:44 +01'00'

Contents

1 Introduction **7**

2 Background **9**
2.1 The Rust programming language . . . 9
2.1.1 Ownership semantics . . . 10
2.1.2 Unsafe Rust . . . 12
2.1.3 Compiler architecture . . . 12
2.2 Lingua Franca . . . 13
2.2.1 Major language components . . . 14
2.2.2 Lingua Franca compiler . . . 18
2.3 Formalisation of the Reactor Model . . . 19
2.3.1 Preliminaries . . . 19
2.3.2 Time . . . 20
2.3.3 Components . . . 20
2.3.4 API for reaction body . . . 22
2.3.5 Reaction precedence . . . 22
2.4 The Rust runtime for Lingua Franca . . . 23
2.4.1 Reaction precedence . . . 23
2.4.2 Main event loop . . . 23
2.4.3 Reaction API . . . 24
2.4.4 Runtime metrics . . . 25

3 Rust software verification **27**
3.1 State-of-the-art verification tools . . . 27
3.1.1 Kani . . . 27
3.1.2 Prusti . . . 31
3.1.3 Creusot . . . 32
3.2 Ownership semantics and verification . . . 34
3.3 Verification with Creusot . . . 35
3.3.1 Verification procedure . . . 35
3.3.2 Writing Creusot specifications . . . 38
3.4 Runtime verification considerations . . . 42
3.4.1 Why Creusot . . . 42
3.4.2 Using Creusot . . . 42
3.4.3 Defining correctness properties . . . 43

4 Properties of a correct runtime **44**
4.1 Termination . . . 44
4.2 Crash freedom . . . 44
4.3 Correctness of VecMap . . . 45
4.4 Formal properties . . . 45
4.4.1 Reactor program state . . . 46
4.4.2 Constant program specification . . . 47
4.4.3 Linear time . . . 47
4.4.4 Reaction precedence derived correctly . . . 47
4.4.5 Event processing . . . 47

5 Verifying the LF Rust runtime **49**
5.1 Verifying the whole runtime . . . 49
5.1.1 Goals . . . 49
5.1.2 Approach . . . 50
5.1.3 Challenges and solutions . . . 51
5.1.4 Conclusion . . . 57
5.2 Verifying VecMap . . . 57
5.2.1 Goals . . . 57
5.2.2 Defining the sorted property . . . 58
5.2.3 Challenges and solutions . . . 58
5.2.4 Reproducing the verification . . . 68
5.2.5 Performance . . . 68
5.2.6 Conclusion . . . 69

6 Conclusion **70**
6.1 Future work . . . 70

1 Introduction

Lingua Franca (LF) is a coordination language for constructing applications in various programming languages, built on the theoretical foundations of the Reactor Model (RM). Its core promise is that of deterministic concurrency. This is achieved through explicit modelling of data dependencies of reactions, the routines of programmer-specified executable code, and discrete-time semantics. The execution of Lingua Franca programs is overseen by a runtime which executes reactions in conformance with the RM. So the promise of deterministic concurrency relies in part on the correct implementation of this runtime. For every programming language that LF supports there is a specific runtime implementation. The latest addition is a runtime implementation for programs written in Rust.
With correctness of implementation being a concern, Rust is already a good choice because it is both memory-safe and guarantees data-race freedom—something that no other LF target language can offer. Additionally, since Rust is compiled ahead-of-time and requires no runtime garbage collection, it rivals the performance of C and C++, which LF can also target.

Rust is still a relatively young programming language, but in recent years interest in formally verifying Rust software has grown in both industry and academia. Formal verification constitutes the logical next step in further guaranteeing the correctness of LF's Rust runtime. Among the available verification tooling, Creusot [DJM22] is the most promising, with a novel approach for modelling pointers [MTK21] that leverages Rust's unique ownership model [Bei16].
The core objective of this work is to define correctness properties for the LF Rust runtime and then to prove their validity in the runtime's codebase using Creusot. Creusot's ability to verify complex Rust software has already been showcased with CreuSAT [Sko22], an optimized SAT solver. That project, however, was built specifically for verification with Creusot. Thus the verification effort of this work will double as a trial as to how applicable Creusot is for previously existing Rust software. The work focuses on surveying the contemporary literature on Rust software verification to give the reader an overview of this emerging field of research and illustrate why Creusot was chosen.

As research into verifying Rust software specifically is only a few years old, the tooling that results from this research is still limited in various ways. These limitations significantly impacted the verification work this thesis set out to do. The encountered limitations and workarounds, if they were found, are documented in this work. One such limitation is Creusot's low coverage of Rust's standard library with contracts it needs for verifying Rust code that uses standard library functionality. As part of the verification effort of this work, Creusot's contract coverage was extended when needed and contributed back to the project.

To summarise, this work surveys three Rust software verification tools, Kani, Prusti, and Creusot, focusing on the latter. It then lays out multiple correctness criteria for the runtime with a justification of why they are crucial for the correct functioning of the runtime. After that, the work details an attempt at verifying the whole runtime according to the defined correctness criteria. However, this was not successful because of Creusot's limitations. The focus then shifts to a module within the runtime that could be decoupled for verification. This module implements `VecMap`, a critical custom data type that is used in multiple places within the runtime. This attempt at verification does succeed, which shows that despite limitations due to a lack of support for certain Rust features, properly isolated code modules that avoid unsupported Rust features can be verified with relatively minor modifications to the original code.

2 Background

In this chapter, we introduce in broad strokes the background for the technologies this thesis examines. For this, it is important to understand how the following concepts interlink. The Reactor Model (RM) underpins the entire work. This theoretical framework is implemented by Lingua Franca (LF), a coordination language that provides the ability to write reactor programs in various programming languages. The semantics in a reactor program is facilitated through a runtime. One language supported by Lingua Franca is Rust, for which there is also a runtime implementation.

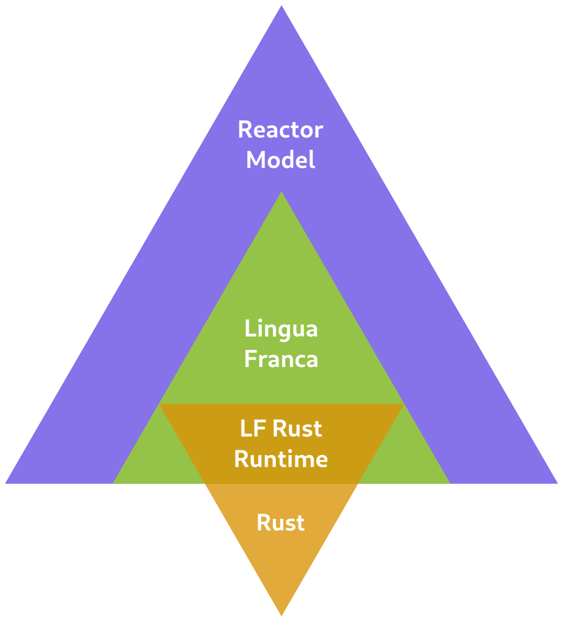


Figure 2.1: Venn diagram showing how the used technologies interact.

2.1 The Rust programming language

Rust is a general-purpose programming language, which had its first stable release in 2015. Since then, it has greatly risen in popularity within industry and academia, now being among the most popular programming languages in 2022 [OGr22].

There are two commonly used release channels of Rust: Stable and Nightly. Stable Rust releases are official and come with backward-compatibility guarantees. Whereas the Nightly release channel includes experimental and yet-to-be-stabilised features [Rus20]. Rust software that relies on such features will only work with Rust Nightly.

2.1.1 Ownership semantics

Rust's core innovation is its ownership semantics. The notion of owned types was not new. It is also present in popular languages like C++. Rust further combines this with strict-aliasing rules for references to owned data. Aliasing occurs when more than one reference to the same memory location is alive within a program [LR91]. For Rust specifically, there are two types of references; shared and exclusive. Shared references may alias but cannot mutate the underlying data, whereas exclusive references provide the inverse functionality. Furthermore, shared and exclusive references to the same owner may not exist simultaneously. The aliasing and mutation rules are illustrated in Table 2.1. These semantics are enforced at compile time, which allows Rust to have a memory-safe data model without garbage collection, as well as guaranteeing the absence of data races [Bei16, §5.2–5.3].

Type	Ownership	Aliasing	Mutation
`T`	owned		✓
`&T`	shared reference	✓	
`&mut T`	exclusive reference[1]		✓

Table 2.1: Overview of aliasing and mutation rules for ownership kinds in Rust. Adapted from [Nik17, at 24:56].

Example

To illustrate the utility of Rust's ownership semantics, let us contrast the C++ example in Listing 2.1 with its Rust counterpart in Listing 2.2. At first glance, the C++ code may appear harmless. However, once we consult the documentation for `push_back`, the problem becomes clear:

> If the new `size()` is greater than `capacity()` then all iterators and references (including the past-the-end iterator) are invalidated. (*cppreference.com* [CPP18])

This means that depending on the state of `v`, pushing the string on line 7 and subsequently printing `ele` may work or fail. If `v` is at capacity, pushing additional contents will reallocate its data and invalidate existing references. When this is the case, `ele` is

[1]These are often also referred to as *mutable* references.

```cpp
1 void strings_ex() {
2     std::vector<std::string> v = {};
3 
4     // other things happen
5 
6     auto& ele = v.front();
7     v.push_back("some string");
8     std::cout << ele << std::endl;
9 }
```

Listing 2.1: C++ example with a potential dangling reference.

```rust
1 fn strings_ex() {
2     let mut v: Vec<String> = vec![];
3 
4     // other things happen
5 
6     let ele: &String = v.first().unwrap();
7     v.push("some string".to_string());
8     println!("{ele}");
9 }
```

Listing 2.2: Direct Rust translation of Listing 2.1 that does not compile.

turned into a dangling reference. In this example, it may cause the printing of incorrect information or a program crash. Additionally, when such a dangling reference is written to, it may further cause memory corruption and security vulnerabilities. Since C++ is not memory safe, this is a valid program, despite the potential error.

The Rust code in Listing 2.2 is the direct counterpart to the C++ code from Listing 2.1. However, due to Rust's strict type system, this is not a valid Rust program. When attempting to compile the example, Rust's compiler will produce the following error: 'cannot borrow `v` as mutable, because it is also borrowed as immutable'.
There is an ownership conflict with `v` here. The type of `ele` is a shared reference (`&String`) borrowing from `v`. And `Vec::push(&mut self, value: T)` requires an exclusive (mutable) reference to `v`. But since a shared reference is currently alive with `ele`, an exclusive reference for mutation cannot be acquired. This conflict triggers Rust's borrow checker to fail the compilation process.

The reader might have noticed that there is a simple way of avoiding the conflict: swapping lines 6 and 7. The exclusive reference that `Vec::push()` requires is dropped when the function call completes. Thus, we can derive other references from `v` again. That version of the program would compile. And if the same change was applied to the C++ version of the program, it would avoid the dangling reference problem.

2.1.2 Unsafe Rust

Rust's memory safety features are sometimes a hindrance. For example, standard Rust does not work with raw pointers (i.e. arbitrary, unchecked memory references including `null`), but their usage might be desired for data structures like trees or performance reasons. To resolve this issue, Rust allows for marking code blocks and functions as `unsafe`. This unlocks Unsafe Rust, which relaxes a few of standard Rust's safety guarantees. As laid out in the book *Programming Rust*:

> An unsafe block unlocks five additional options for you:
>
> - You can call `unsafe` functions. […]
> - You can dereference raw pointers. Safe code can pass raw pointers around, compare them, and create them by conversion from references (or even from integers), but only unsafe code can actually use them to access memory. […]
> - You can access the fields of unions, which the compiler can't be sure contain valid bit patterns for their respective types.
> - You can access mutable static variables. […] Rust can't be sure when threads are using mutable static variables, so their contract requires you to ensure all access is properly synchronized.
> - You can access functions and variables declared through Rust's foreign function interface. These are considered unsafe even when immutable since they are visible to code written in other languages that may not respect Rust's safety rules.
>
> [BOT21, p. 628]

2.1.3 Compiler architecture

Having a high-level understanding of Rust's compilation process is essential because many state-of-the-art verification tools extract information from the process to facilitate their functionality. Broadly, the compilation process is divided into four stages. The following descriptions were adapted from the *Guide to Rustc Development* [Rus22].

Lexing & parsing Technically, these are two stages combined into one for brevity here. The source code is transformed into an Abstract Syntax Tree (AST) for further processing.

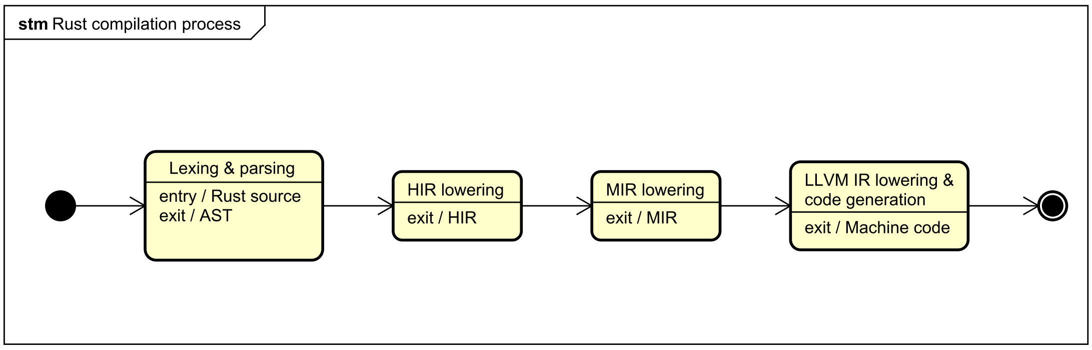


Figure 2.2: Illustration of Rust's compilation process.

High-Level Intermediate Representation (HIR) lowering This IR still closely resembles the original Rust code. Certain language constructs like loops are desugared. Furthermore, type inference and type checking happen at this stage.

Mid-level Intermediate Representation (MIR) lowering This stage validates Rust's ownership semantics (also called borrow checking). MIR represents a control flow graph on which optimisations with retained Rust semantics are performed.

LLVM IR lowering & code generation Again, technically these are two stages compressed into one for brevity. Rust uses LLVM as its code generation backend for the many hardware platforms it supports. To do this, MIR is lowered to LLVM IR.

2.2 Lingua Franca

The theoretical foundation of Lingua Franca is the Reactor Model. In a nutshell, reactors are reactive program components whose behaviour and interactions enjoy deterministic, time-based semantics.
Reactions form the atomic computational unit of the RM. They are routines grouped by reactors over which they share common state. As the name implies, reactions are executed in reaction to certain events. Each reaction explicitly declares its trigger events and other data dependencies required for execution. The dependency information is then used to determine an execution plan of reactions that enable the model's determinism. Dependencies alone are not enough for this, however. Since reactions share mutable state across their reactions, competition over those resources has to be resolved deterministically. The model does this by defining a total order for reactions within a reactor and scheduling reaction execution along a well-defined logical timeline. A tag is an instant on this logical timeline. If two reactions are triggered at the same tag, they are executed according to the total reaction order. Shared state is isolated to reactors, which easily allows for concurrency within the model. Reactions from different reactors that have

```
target Rust;

main reactor GreetingsAndFarewell {
    reaction (startup) {=
        println!("Hello!");
    =}

    reaction (shutdown) {=
        println!("Goodbye!");
    =}
}
```

Listing 2.3: LF example program containing two reactions.

their trigger satisfied can be executed in parallel without hurting the RM's promise of determinism.
However, if desired, non-determinism can be introduced deliberately to reactor programs via physical actions. They are designed to allow outside input (e.g. a button press) into a reactor program. The key benefit is that, while physical actions may be scheduled non-deterministically, the rest of the program retains the RM's promise of determinism.

Lingua Franca, an implementation of the RM, is a coordination language that requires writing the reaction logic in a specific programming language. All other RM components and their relations are expressed in LF's syntax. During the compilation process, this code is transformed fully into the target language. The generated target code makes use of a runtime library to facilitate the RM semantics. As of the time of writing, supported LF target languages are: C, C++, Python, TypeScript, and Rust.

Every LF source file has to declare its target language. And each LF program has to declare a main reactor, which represents the top-most reactor in the reactor topology (i.e. it transitively contains all other reactors).
Take, for example, the LF program in Listing 2.3. First, it declares the program's target language: Rust. The main reactor called `GreetingsAndFarewell` contains two reactions triggered on program startup and shutdown respectively. The body of reactions enclosed by `{=` and `=}` is written in the target language. When executed, the program will print 'Hello!', as that is triggered by startup, as defined. Then the program has no further events to process and initiates shutdown automatically. This triggers the reaction that prints 'Goodbye!' and after that, finally, the program terminates.

2.2.1 Major language components

This section explains the major Lingua Franca language components.

```
target Rust;

reactor Square (x: f64(0.0)) {
    state x: f64(x);

    reaction (startup) {=
        println!("The square has an area of {}.", self.x * self.x);
    =}
}

main reactor Main {
    square = new Square(x = 3.5);
}
```

Listing 2.4: Example of one reactor containing another in LF.

Reactors

Each reactor has a set of private state variables and a set of reactions. The internal state variables of the reactor can be shared across reactions when their data dependencies are declared appropriately. One reactor may contain several others.
For an example of reactor containment, see Listing 2.4. Here the reactor `Main` contains `Square`. This reactor has one state variable denoting the side length of the square it represents. And further, the internal state is used in the only reaction to print the square's area.

Reactions

Reactions take the general form

`reaction(<triggers>) <sources> -> <effects> {= … =}`.

Each block enclosed by `<` and `>` represent comma-separated lists of reactor components. In detail they denote the following:

Triggers The components that, when present, trigger the reaction. The triggering occurs if one or more of the listed triggers is present. A reaction requires at least one trigger.

Sources The reactor components the reaction gets access to in a read-only form. Its declaration is optional.

Effects The list of reactor components the reaction can write to. Its declaration is optional, but if omitted, `->` must be omitted as well.

Event sources

Event sources produce events that reactions may be triggered by. Previous examples, like Listing 2.3, already introduced two special events: `startup` and `shutdown`. As their name implies, these events are produced at program startup and shutdown respectively. But LF contains other constructs that can serve as sources for events. Among them are ports and actions.

Ports Ports are an important language construct for reactor communication. They facilitate the only way for reactors to communicate with each other. A port is declared as either input or output and associated with a specific reactor. A reactor has access to the ports of the reactors it immediately contains[2], but, crucially, nothing else. Accessible ports can be connected—output to input. When a reaction writes to an output port, the written data is propagated to the connected inputs logically simultaneously. This means that reactions depending on such input as a trigger get executed at the same tag as the reaction that originally wrote to the output. In order to prevent port connections from blocking the advancing of logical time, port connections and the reactions that depend on them and set them are forbidden from forming a cycle.

Actions Actions are exclusively for triggering reactions within their reactor. They are scheduled during reaction execution and are split into logical and physical actions. Logical actions are scheduled synchronously, whereas for physical actions the scheduling is asynchronous. This allows the programmer to use physical actions to deliberately introduce non-determinism into the program. For example, a physical action could be used to trigger a reaction on user input. If a certain button is pressed, a LF program could schedule a physical action that alerts all reactions depending on that action as a trigger. But the asynchronicity is exclusive to the scheduled physical actions. Reactions triggered by physical actions are executed synchronously.

Example

The example LF program in Listing 2.5 showcases all the introduced language components. It computes the n^{th} entry in the Fibonacci sequence, as given by `seq_idx` on line 40 which is a parameter of the main reactor `Fibonacci`. The main reactor contains two further reactors. `Adder` takes in two summands and writes their sum to the reactor's output. And `Holder` holds the current intermediate step of the computation of the Fibonacci sequence and further tracks how many computational steps remain to obtain the result.
`Holder` uses a logical action to advance the sequence computation. When receiving the next value in the sequence on input `next_f`, it adjusts its internal state with this new

[2]There is no such access through transitive containment.

```
target Rust;

reactor Adder {
    input a: u64;
    input b: u64;
    output sum: u64;

    reaction(a, b) -> sum {=
        if ctx.is_present(a) && ctx.is_present(b) {
            ctx.set(sum, ctx.get(a).unwrap() + ctx.get(b).unwrap());
        }
    =}
}

reactor Holder(rounds: usize(1)) {
    state rounds: usize(rounds);
    state f_n: u64(0);
    state f_m: u64(1);
    logical action next;
    input next_f: u64;
    output a: u64;
    output b: u64;
    output result: u64;

    reaction(startup, next) -> a, b, result {=
        if self.rounds > 0 {
            ctx.set(a, self.f_n);
            ctx.set(b, self.f_m);
            self.rounds -= 1;
        } else { ctx.set(result, self.f_n); }
    =}

    reaction(next_f) -> next {=
        self.f_n = self.f_m;
        self.f_m = ctx.get(next_f).unwrap();
        ctx.schedule(next, Asap);
    =}
}

main reactor Fibonacci(seq_idx: usize(1)) {
    state seq_idx: usize(seq_idx);
    holder = new Holder(rounds=seq_idx);
    adder = new Adder();
    holder.a -> adder.a;
    holder.b -> adder.b;
    adder.sum -> holder.next_f;

    reaction(holder.result) {=
        let result = ctx.get(holder__result).unwrap();
        println!("In the Fibonacci sequence {} is at position {}.", result, self.seq_idx);
    =}
}
```

Listing 2.5: A reactor program computing a number in the Fibonacci sequence. It showcases multiple reactors, ports, port connections, and logical actions.

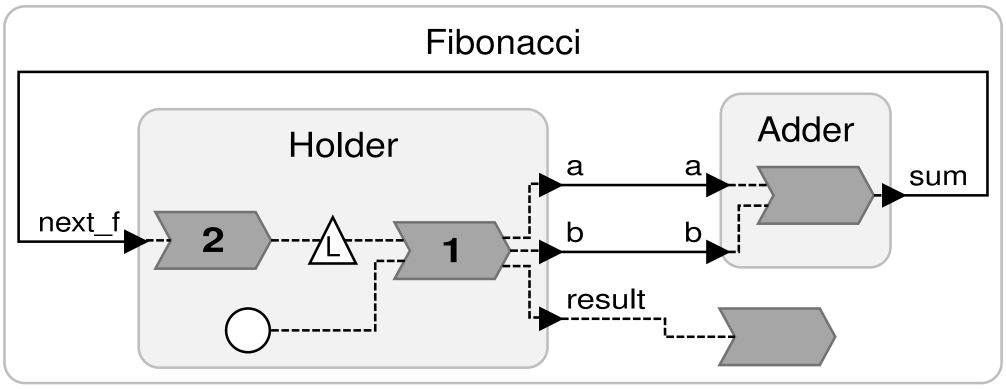


Figure 2.3: A Diagram illustrating the structure of the reactor program in Listing 2.5. Reactors are represented by rounded quadrilaterals. Their reactions are represented by arrow-like shapes. If a reactor contains more than two reactions, the numbers printed on top of the reactions represent their rank in the total order of reactions within that reactor. The triangle shapes represent actions (L for logical). The circle in `Holder` represents the startup event. And lastly, the arrowheads at the edges of reactors represent their input and output ports.

value and schedules the logical action `next`. This then triggers the reaction depending on it, which puts the last two values in the sequence on the output and awaits the next sequence value, until the result is reached. The main reactor composes `Adder` and `Holder` so that `Adder` computes the next value for `Holder`. This is achieved by connecting the right ports from these reactors. The example further demonstrates, how reactions can directly depend on the output ports of contained reactors. This can be seen in the only reaction of `Fibonacci`. How all of these components overall fit together is illustrated in Figure 2.3.

2.2.2 Lingua Franca compiler

The Lingua Franca compiler (LFC) is the program that transforms a complete LF program to the final executable. The form that these executables take, depends on the target language of the LF sources. But for the purposes of this thesis, we will limit our view to the compilation process for the Rust target.

LFC can broadly be divided into a frontend and a backend. The frontend is shared among targets and has the task of parsing the LF sources into an abstract syntax tree (AST) for further processing in the backend. In the backend, the specific code generator for the target in question is invoked. These transform the reactor structure described in LF to native constructs of the target language. The reaction bodies, which are already written in the target language, are inserted by the generator. Here reactions and the

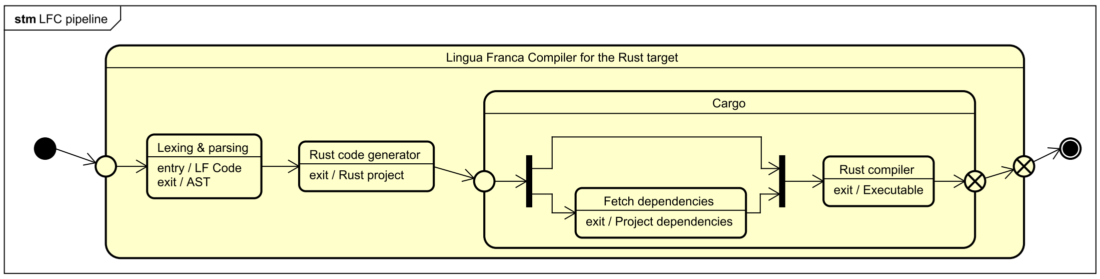


Figure 2.4: Illustration of the compilation process of LFC for the Rust target.

other reactor model constructs are composed with the runtime library. Ultimately, the generator produces a complete Rust program, which is then handed over for compilation. The compilation is managed by the Cargo utility, which provides package management and compilation functionality. Cargo is invoked on the generated source, fetches its dependencies and compiles it all to a single executable file.

2.3 Formalisation of the Reactor Model

The first formal description of the Reactor Model was offered in *Reactors: A Deterministic Model for Composable Reactive Systems* [Loh+20], which was then refined in [Loh20]. This section mostly follows what was laid out in those works and highlights where LF diverges. Formal descriptions of the RM usually define mutations, constructs similar to reactions that can alter the structure of reactors that contain them, but so far Lingua Franca has not implemented them. For this reason, they will be omitted here. Another such concept is reaction deadlines, which have not been implemented in the LF Rust runtime. Thus it will be omitted as well.

2.3.1 Preliminaries

Most formalised components in this section will be described as tuples. Since the RM heavily relies on components interacting with each other, like in the reference literature, we define a shorthand for accessing individual tuple elements like so:

Definition 2.3.1. Let $t = (a, b)$ be an instance of the tuple construct described by (A, B). Then A also describes a mapping such that $A(t) = a$. The mapping B is defined analogously.

The following definitions will make heavy use of two sets:

Identifiers (Σ) Used to uniquely refer to various components.

Values (V) Refers to the values used within and exchanged with other reactors. The values are untyped. There is a special value denoting when values are absent, $\epsilon \in V$.

2.3.2 Time

A time instance in the RM is represented by a tag from the set $\mathbb{G}$, where $\mathbb{G} = \mathbb{T} \times \mathbb{N}$. The first part of a tag refers to a discrete-time value. The exact definition of $\mathbb{T}$ will depend on the implementation. By convention, $\mathbb{N}$ is the set of natural numbers. It describes the micro-step index to the time value it is associated with.

Definition 2.3.2 (Tag order)**.** Elements of $\mathbb{G}$ are totally ordered lexicographically, i.e.

$$\forall (t, m), (t', m') \in \mathbb{G} : (t, m) < (t', m') \iff (t < t') \vee (t = t' \wedge m < m').$$

Definition 2.3.3 (Logical time)**.** Logical time is a monotonically increasing sequence of tags in $\mathbb{G}$. When reactions execute at the same tag, they are said to be logically simultaneous.

Definition 2.3.4 (Physical time)**.** Physical time is a monotonically non-decreasing sequence of integer values obtained from a device for measuring time (e.g. a clock).

2.3.3 Components

Definition 2.3.5 (Events)**.** An event e is a tuple $e = (t, v, g)$ where

- $t \in \Sigma$ is the event's trigger,
- $v \in V$,
- $g \in \mathbb{G}$ is the event's assigned tag.

Definition 2.3.6 (Reactors)**.** A reactor r is a tuple $r = (I, O, A, S, N, R, P, G, \{\bullet, \diamond\})$ where

- $I \subseteq \Sigma$ is the set of inputs,
- $O \subseteq \Sigma$ is the set of outputs,
- $A \subseteq A \times \{\text{logical}, \text{physical}\}$ is the set of actions,
- $S \subseteq \Sigma$ is the set of state variables,
- N is the set of reactions,
- R is the set of contained reactors,

- $P : N \to \mathbb{P}$ is the priority function,
- $G \subseteq \bigcup_{r \in R} O(r) \times \bigcup_{r \in R} I(r)$ is the topology graph,
- $\{\bullet, \diamond\} \subseteq \Sigma$ are the special triggers for startup and shutdown respectively.

All identifiers used in a reactor and its contained reactors are unique. P assigns an order to the reactor's reactions which is used for deterministically resolving competition over resources when two reactions are triggered at the same tag. The order is defined by mapping reactions to $\mathbb{P}$ (defined below), the priority set. The topology graph G describes how inputs and outputs of contained reactors are connected. When a value becomes available on an output, it is automatically and logically simultaneously propagated downstream across the edge in G to the designated input.

Definition 2.3.7 (Priority set $\mathbb{P}$)**.** The priority set is given by $\mathbb{P} = \mathbb{N}_+ \cup \{*\}$, where $\mathbb{N}_+ = \mathbb{N} \setminus 0$. $\mathbb{P}$ has a partial order given by the order in $\mathbb{N}_+$, extended with the following rule for $*$:

$$* \leq * \land \forall n \in \mathbb{N}_+ : n < *.$$

Reactions that do not access the reactor state are assigned priority $*$. Since those reactions definitionally do not compete for resources they do not need to be assigned a priority. This means that reactions with priority $*$ are easily parallelizable.

There are some crucial differences between this formalisation and the implementation in Lingua Franca:

- G encodes the port connections that a reactor can declare for its contained reactors' ports. But in LF, port connections may also be declared from and to the reactor's own ports, which this formalisation does not account for.
- In this formalisation $\{\bullet, \diamond\}$ are not more concretely defined. With this, it would be possible to have individual startup and shutdown triggers for each reactor. In LF however, startup and shutdown are unique global triggers that are the same for each reactor.

Definition 2.3.8 (Reactions)**.** A reaction n is a tuple $n = (D, T, B, D^\vee, H)$, where

- $D \subseteq I(C(n)) \cup \bigcup_{r \in R(C(n))} O(r)$ is the set of dependencies,
- $T \subseteq D \cup A(C(n)) \cup \{\bullet, \diamond\}$ is the set of triggers,
- B is the body of the reaction (i.e. the executable code of the reaction),
- $D^\vee \subseteq O(C(n)) \cup \bigcup_{r \in R(C(n))} I(r)$ is the set of anti-dependencies,
- $H \subseteq A(C(n))$ is the set of schedulable actions.

C is the containment function for reactions.

Definition 2.3.9. The containment function C maps reactions to the reactors that contain them. Let n be a reaction, then $C(n) = r$ s.t. $n \in N(r)$.

The sets D, T, and $D^{\vee}$ relate to the concept of triggers, sources, and effects for reactions in LF as introduced in §2.2.1. Triggers in LF and T in this formalisation are equivalent. The same is true of effects and $D^{\vee}$. But for sources, it is equivalent to $D \setminus T$.

2.3.4 API for reaction body

The following procedures are available in the body of reactions to affect the execution of the reactor program.

currentTag Get the current logical time.

physicalTime Get the current physical time.

get Returns the value currently associated with the specified port or action from D.

set Binds the specified value to the specified port in $D^{\vee}$.

schedule Schedule a given action in H with a minimum delay of one micro-step.

requestStop Issue request to runtime for the reactor program to terminate.

2.3.5 Reaction precedence

In the same way that reactions within a reactor have a well-defined precedence through P, the RM requires such an ordering on all reactions for execution precedence to ensure determinism.

Definition 2.3.10. Let $\prec$ be the binary reaction precedence relation. For two reactions n_1, n_2, $n_1 \prec n_2$ (i.e. n_1 is of higher precedence than n_2) if n_2 has a dependency on an anti-dependency of n_1. Additionally, for two reactions n_3, n_4 within the same reactor:

$$P(n_3) < P(n_4) \implies n_3 \prec n_4$$

2.4 The Rust runtime for Lingua Franca

The Rust LF runtime library provides several critical components for reactor programs written in Rust. A first working implementation was finalised during Clément Fournier's thesis work *A Rust Backend for Lingua Franca* [Fou21] in 2021[3].
The runtime includes the startup routine, the main event processing loop, and the event scheduler. On top of that, it contains static routines (e.g. the connecting of ports) that the LFC code generator will use in assembling the reactor program. When discussing specificities of the implementation, it will be in reference to the version of Git commit hash `98991f3`.

2.4.1 Reaction precedence

The Rust runtime follows the routine outlined in [Loh+20, §3.6]. When the reactor program starts up, it builds a dependency graph for reactions which is a representation of the precedence relation. But since traversing a graph is too expensive at runtime, the graph is transformed into a set of so-called levels.

Definition 2.4.1 (Level [Fou21])**.** Given a directed acyclic graph (DAG), the level of a vertex is the length of the longest path from any root of the graph to that vertex.

Since the level sets of reactions are directly derived from the dependency graph, reactions of different levels may have a dependency between them, but reactions of the same level are by construction independent [Fou21, p. 39].

2.4.2 Main event loop

The Rust runtime operates on a loop where it takes the closest upcoming reactions from the event queue, processes them, stores their effects, and then repeats until the shutdown is initiated. Physical actions complicate this simple loop, so for explanation purposes they are omitted here.

Event and reaction queue

Crucially, the runtime's event queue does not contain events, but instead, it stores the reactions triggered by events for a specific tag in lists sorted by their level. On each iteration, the upcoming event's triggered reactions are removed from the queue and given to the `process_tag` routine, which works similarly to how it is defined in [Loh+20, p. 17].
The reaction queue stores triggered reactions for the tag that is currently being processed.

[3]The current implementation can be found at `https://github.com/lf-lang/reactor-rs`.

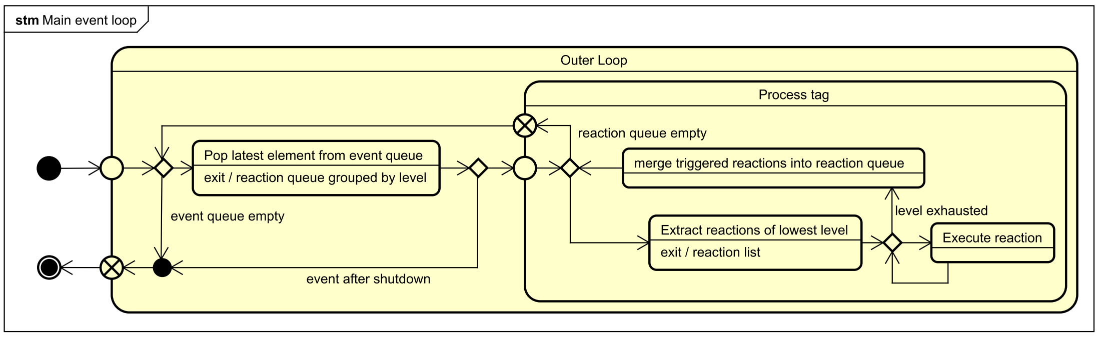

stm Main event loop
Outer Loop
Process tag
reaction queue empty
Pop latest element from event queue
exit / reaction queue grouped by level
merge triggered reactions into reaction queue
level exhausted
event queue empty
Extract reactions of lowest level
exit / reaction list
Execute reaction
event after shutdown

Figure 2.5: UML state machine diagram of the main event loop.

These reactions will exclusively be the ones triggered by dependencies on ports since data on outputs is propagated to connected inputs instantaneously w.r.t. logical time. Since actions are scheduled with minimally one micro-step of delay, those reactions exclusively triggered by them will not be in the reaction queue.

Processing a tag

When `process_tag` is invoked with the initial reactions for the tag, the routine extracts the reactions of the lowest level (i.e. with the highest precedence). These are then executed one after the other, or, if parallelism is desired, executed in parallel. Scheduled actions are already added to the event queue during execution, while immediately triggered reactions are merged into the reaction queue after the processing of a level finishes. And then the processing loop restarts at extracting the reactions of the lowest level on the reaction queue. This is repeated until the reaction queue is fully drained. Figure 2.5 illustrates the whole scheduler main event loop, including the `process_tag` routine.

2.4.3 Reaction API

The Rust LF runtime provides the API as defined in §2.3.4 in reaction bodies. It is provided through the `ReactionCtx` object, available in reaction bodies through the `ctx` variable. It also provides additional functionality (e.g. how much time has elapsed since execution start). Details are documented in the runtime's API documentation [Rusb].

2.4.4 Runtime metrics

As measured with the code statistics tool Tokei[4][5], the runtime consists of 5879 lines of code, of which 826 are comments and 849 are blanks. But since Rust allows for easy usage of dependencies with its package manager Cargo, the amount of code actually compiled is much greater. Figure 2.7 shows the dependency graph of the Rust LF runtime.

Runtime performance

The Rust runtime performs well compared to other LF targets. Of primary interest is the comparison with C and C++, since they also are ahead-of-time compiled systems programming languages. Against them, the Rust target performs competitively in benchmarks from LF's benchmarking suite[6], as can be seen in Figure 2.6[7]. This performance is the product of the optimisation work detailed in [Fou21, §5.1]. Some of these optimisations rely on using Unsafe Rust.

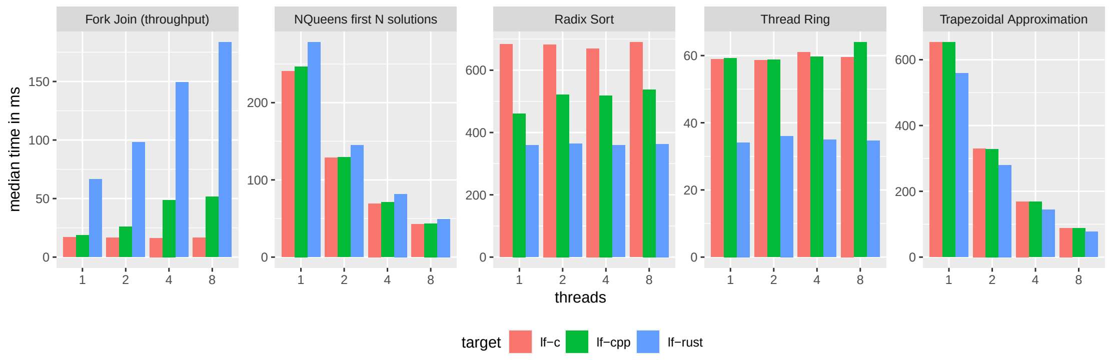


Figure 2.6: Five benchmark result comparisons between LF's C, C++, and Rust targets. The *Fork Join (throughput)* benchmark highlights a performance problem within the Rust runtime, especially with parallel execution. But otherwise, the performance of the Rust target is very competitive or significantly better than those of C and C++.

[4]The tool is available from `https://github.com/XAMPPRocky/tokei`.

[5]The version used is 12.1.2.

[6]The benchmarking suite is available at `https://github.com/lf-lang/benchmarks-lingua-franca`.

[7]Reproducibility: Benchmarks performed on a machine with an Intel(R) Core(TM) i9-10900K CPU with 32 GB of RAM. The version of LF is the one identified by Git commit `7d52516` and for the benchmarking suite it is `dfba277`.

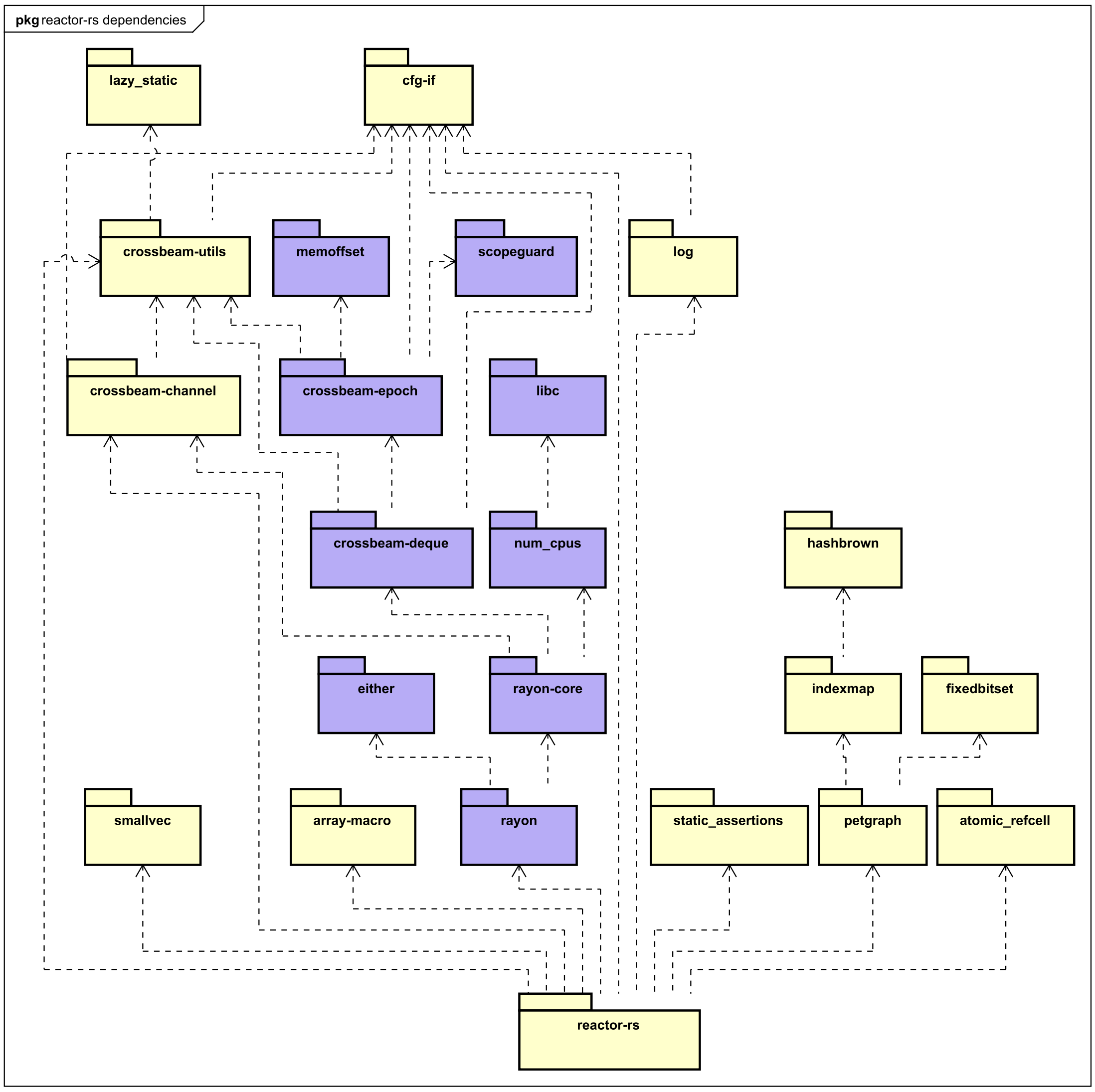


Figure 2.7: Full dependency graph of the Rust LF runtime. The dependencies purely used for parallel execution are coloured purple.

3 Rust software verification

This chapter introduces several state-of-the-art verification tools for Rust software: Kani, Prusti, and Creusot. We will put extra emphasis on Creusot since it is the tool we elected to be the most promising at the start of this work. Further, we will explain how Rust's ownership model can help in verifying pointer-based programs.

3.1 State-of-the-art verification tools

Attempts at verifying Rust software are almost as old as the language itself. Initial work used the LLVM IR of the compilation process as a foundation (e.g. [LAL18] and [BHR18]), for which tooling was already available. Over the years academic interest in Rust surged and as a result, approaches to verifying Rust software grew more sophisticated. State-of-the-art verification tools all use MIR, the intermediate representation used before lowering to LLVM IR. This is because MIR preserves high-level abstractions, including Rust's unique ownership semantics. For details on Rust's compilation process see §2.1.3.

To illustrate the functionality of each presented tool, we will use the running example given in Listing 3.1. The property we wish to verify is the one outlined in the comment above the function: *assuming the provided vector is sorted, it remains sorted after calling `insert_in_order()`.* We will outline how this property can be specified with each tool in the following subsections.

3.1.1 Kani

Kani is a model checker for Rust programs. The tool is publicly developed as open source software[1] at Amazon Web Services (AWS). It has been used to verify properties of AWS's Firecracker virtual machine management solution [Van+22], for example. Under the hood, Kani employs the CBMC model checker [CKL04] for the C programming language, by converting MIR to C with `goto`-statements. Further, it extracts information about traits from the Rust program, to gain additional function pointer restrictions for the generated C code to reduce verification time [Van+22, §3.3]. By default, Kani checks for memory unsoundness issues (e.g. out-of-bounds memory access), but the user may add

[1]The project can be found at `https://github.com/model-checking/kani`.

```rust
/// Insert item into v. This preserves ordering of elements,
/// assuming they were ordered in the first place.
fn insert_in_order<T: Ord>(v: &mut Vec<T>, item: T) {
    let mut insert_idx = 0;
    if v.last() <= Some(&item) {
        insert_idx = v.len();
    } else {
        for idx in 0..v.len() {
            if v[idx] > item {
                insert_idx = idx;
                break;
            }
        }
    }
    v.insert(insert_idx, item);
}
```

Listing 3.1: This is an insert function over the generic type `T`. We provide the vector `v` as a mutable reference to be inserted into and an `item` for insertion. The trait `Ord` which `T` implements, requires that there is a total order over the instances of the type. This allows us to compare the ordering of elements despite `T` being generic. The function itself finds an appropriate insertion index (`insert_idx`), under the assumption that the vector is sorted: if a last element in `v` exists and is greater than the item we want to insert, we insert it at the end, otherwise, we linearly search the vector for the right point.

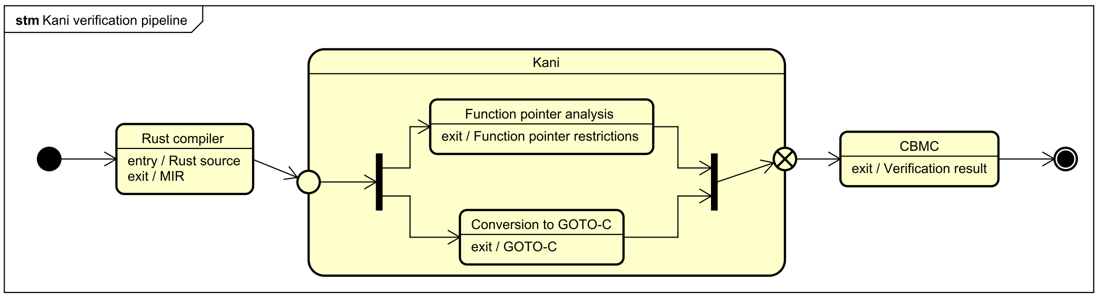

stm Kani verification pipeline
Rust compiler
entry / Rust source
exit / MIR
Kani
Function pointer analysis
exit / Function pointer restrictions
Conversion to GOTO-C
exit / GOTO-C
CBMC
exit / Verification result

Figure 3.1: An illustration of Kani's verification pipeline.

additional arbitrary properties to verify in a Kani proof. Currently, the model checker does not support any of Rust's concurrency constructs [Van+22, §3.1.1].

Considering the running example, Listing 3.2 shows how the order-preserving property can be verified. Notice how a function `is_sorted()` was added to verify the property before and after. Notice also, how the actual proof code in `verify()` is a separate function which applies `insert_in_order()` to a vector of `u64` integer types. It does not verify the generic case in which the function is defined. This is because Kani is a model checker. As defined by Baier and Katoen:

> Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model. [BK08, p. 11]

So Kani needs an explicit program to operate on—it cannot abstractly reason about code. We might inductively reason that since the ordering property holds for integer types like `u64`, it will probably also hold for other types with a total order. But Kani cannot do this reasoning for us, it can only tell us that the checked properties are valid in this particular instance. Further, our inductive reasoning may not prove true. There are many reasons why `insert_in_order()` could have the order-preserving property for `u64`, but not for other types. What if the function relied on an assumption about how many different values the generic type can have?

There is another problem. Notice how in `verify()` the size of the generated vector `v` is set as variable `X`. When running the Kani tool this would have to be substituted with a fixed number. Then `kani::any()` will produce slices of that size or lesser, filled with arbitrary instances of `u64`. Then why not set `X` to the maximum size of a vector? Because this would produce a state space so large, that verification on contemporary hardware would be impossible. So we can only verify the property over a subset of the possible input. But even with these restrictions, we run into issues quickly. If $\texttt{X} = 0$, there is one possible value for `v`: the empty vector. If $\texttt{X} = 1$ there are 2^{64} possible values for `v`, because that is how many different valuations of `u64` there are. The problem gets exponentially worse from here. Table 3.1 shows the exponential increase in verification

```rust
fn is_sorted<T: Ord>(slice: &[T]) -> bool {
    for idx in 1..slice.len() {
        if slice[idx - 1] > slice[idx] {
            return false;
        }
    }
    true
}

#[cfg(kani)]
#[kani::proof]
fn verify() {
    let mut v = kani::any::<[u64; X]>().to_vec();
    kani::assume(is_sorted(&v));
    let item: u64 = kani::any();
    insert_in_order(&mut v, item);
    assert!(is_sorted(&v));
}
```

Listing 3.2: The Kani proof for the function from Listing 3.1 (omitted here). `is_sorted()` defines the property we wish to check before calling the function and after. The proof itself is in `verify()`. `kani::any()` is used to produce an arbitrary instance of a given type. For example, we use it to produce an arbitrary unsigned 64-bit integer for `item`. And we use it to create arbitrary slices of that type up to a given length (written as `X`) and convert them to a vector for `v`. `kani::assume()` tells the model checker to discard all states in which the contained property is not satisfied. Finally, Kani overwrites the standard `assert!()` macro, to verify that the contained property is satisfied for every state.

`X`	Kani verification time in seconds
0	0.38
1	0.88
5	6.55
10	24.21
20	102.49
30	257.09

Table 3.1: The verification times on the running example with varying `X`. The verification times blow up exponentially very quickly.

time with linear increases of `X`[2]. Verification of the order-preserving property for vectors of any size is not realistic with Kani.

Admittedly, our running example is particularly ill-suited for verification with Kani. But it will serve well in highlighting the differences it has with Prusti and Creusot.

3.1.2 Prusti

Prusti [Ast+22] is a deductive verification tool for Rust software. Under the hood, it utilises the Viper verification platform [MSS16]. Instead of solely relying on the Rust compiler for verifying the ownership model in Rust code, Prusti models Rust's memory model separately for verification. By default, Prusti checks for the unreachability of panics and the absence of integer overflows in Rust code. Additionally, custom properties may be verified by annotating code with, for example, pre- and postconditions. However, the project still appears to be quite immature. Several Rust language features used in the running example, like the `Vec` type and `for`-loops (due to their reliance on iterators) are at the time of writing not supported by Prusti. Even with adjustments to the running example to work around these issues, no verification of the absence of panics and integer overflows was possible due to the tool crashing.

This is unfortunate because Prusti's approach of deductive verification is much more promising for the running example. Instead of model checking, where the program is modelled as a set of states and transitions from one to the other, deductive verification formally abstracts the programs so that we can deductively reason about their functionality. Since the reasoning is over an abstract model, instead of concrete program states, the size of the vector in the running example or the generic definition of the function is not a problem like it is with Kani. If Prusti was more mature, we would be able to directly annotate the running example's function with an expression of the order-preserving property and verify its correctness through Viper.

[2]The numbers were obtained by running Kani version 0.22.0 on a computer with an Intel(R) Core(TM) i9-10900K CPU and 32 GB of RAM.

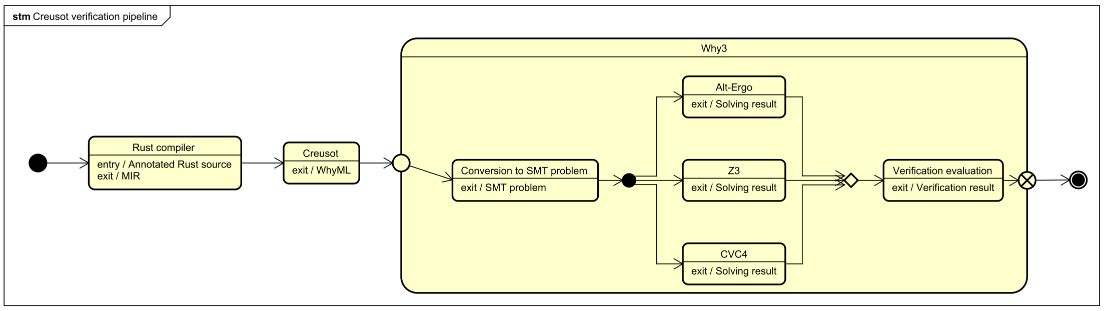


Figure 3.2: An illustration of Creusot's verification pipeline.

3.1.3 Creusot

For an in-depth description of Creusot's features, see §3.3.

Creusot [DJM22] is a deductive verification tool for Rust software. It is developed publicly as free software[3]. Properties can be defined in Rust code through annotations and special logic functions. The tool transforms MIR to WhyML, the ML dialect of the Why3 proof environment [Bob+11]. For verification, the generated WhyML code is loaded into Why3, where individual proofs are attempted to be verified through external SMT solvers like Alt-Ergo [Con+18], CVC4 [Bar+11], and Z3 [MB08].
By default, specifications generated by Creusot check for crash freedom of the software it was derived from. However, at the time of writing, the tool does not support concurrent or unsafe Rust code. One of Creusot's major achievements that underscore its utility is *CreuSAT*, a verified SAT solver written in Rust [Sko22].

Listing 3.3 gives the specification of the running example in Creusot. Note how the order-preserving property was expressed in a precondition and postcondition in lines 3 and 4. The precondition requires that `v` is sorted and the postcondition ensures that `v` is still sorted at the end of the function. Note further, how this specification requires Creusot-specific trait constraints. The last annotation is the invariant on the `for`-loop. The annotation shows off several features of Pearlite, Creusot's language for logic expressions, like implication and universal quantification. The invariant is crucial for verification because it specifies that up until the current index in the iteration, all items before that point in `v` are less than or equal to `item`. `produced` is an internal variable Creusot provides when going through an iterator in a `for`-loop. It holds a sequence of all produced values of the iterator up until that point in the iteration. Lastly, in the first line, we import everything from Creusot's contracts library. It contains the specifications for Creusot-internal types and for standard library contents. For example, the specification of `Vec::insert()`, which is used on the last line of the function can be seen in Listing 3.4.

[3]The project can be found at https://github.com/xldenis/creusot.

```rust
use creusot_contracts::*;

#[requires(v.deep_model().sorted())]
#[ensures((^v).deep_model().sorted())]
fn insert_in_order<T>(v: &mut Vec<T>, item: T)
where
    T: Ord + DeepModel,
    T::DeepModelTy: OrdLogic,
{
    let mut insert_idx = 0;
    if v.last() <= Some(&item) {
        insert_idx = v.len();
    } else {
        #[invariant(prev_leq, forall<i: Int> i >= 0 && i < produced.len() ==>
                    v.deep_model()[i] <= item.deep_model())]
        for idx in 0..v.len() {
            if v[idx] > item {
                insert_idx = idx;
                break;
            }
        }
    }
    v.insert(insert_idx, item);
}
```

Listing 3.3: The Creusot specification for the function from Listing 3.1. This version imports Creusot's contracts library at the top. The new annotations contain the specification for the function. `requires()` describes a precondition, whereas `ensures()` is a postcondition. `invariant()` contains the invariant for the loop below. It is expected to be true at every iteration start of the loop. Lastly, there are additional trait constraints on `T` which are specific to Creusot.

```rust
#[ensures((@^self).len() == (@self).len() + 1)]
#[ensures(forall<i: Int> 0 <= i && i < @index ==>
         (@^self)[i] == (@self)[i])]
#[ensures((@^self)[@index] == element)]
#[ensures(forall<i: Int> @index < i && i < (@^self).len() ==>
         (@^self)[i] == (@self)[i - 1])]
fn insert(&mut self, index: usize, element: T);
```

Listing 3.4: The Creusot specification for `Vec::insert()`.

When running Creusot on this example, it produces WhyML output combining the specification and the translated code. This file can then be loaded into Why3. And there running the verification process at the highest level of automation produces a proof for this specification.

3.2 Ownership semantics and verification

The core abstraction that is commonly used today for verifying programs that use pointers is separation logic. The memory of the program and its actions on it (allocation, deallocation, and read/write access) are modelled separately. One of the foundational publications on separation logic by O'Hearn et al. summarises the problem it solves:

> Pointers have been a persistent trouble area in program proving. The main difficulty is not one of finding an in-principle adequate axiomatization of pointer operations; rather there is a mismatch between simple intuitions about the way that pointer operations work and the complexity of their axiomatic treatments. For example, pointer assignment is operationally simple, but when there is aliasing, arising from several pointers to a given cell, then an alteration to that cell may affect the values of many syntactically unrelated expressions. [ORY01]

Crucially, with Rust's ownership semantics, this problem does not arise since aliasing and data mutation are mutually exclusive. This makes the separation logic approach superfluous for Rust. *RustHorn* [MTK21] describes a technique for transforming a safe subset of Rust into constrained Horn clauses, a fragment of first-order logic. This approach supersedes the requirement of separately modelling the program's memory. Instead, it represents a mutable reference `&mut ma` as a tuple $(a, a_\circ)$, where a represents the current value of `*ma` (`ma` dereferenced) and $a_\circ$ represents the future value of `*ma` at the moment the reference dies. Having $a_\circ$ in this abstraction is important because when the reference dies, the underlying data may be aliased again. Later, *RustHornBelt* [Mat+22] utilised previous work, *RustBelt* [Jun+17], to semantically prove the validity of the *RustHorn* approach to verifying Rust programs.
Creusot, presented in §3.1.3, utilises the insight from *RustHorn* for its transformation from MIR to WhyML. Within its annotations, the user can refer to the final value of a mutable reference `&mut ma` through the final operator `^ma`.

One limiting factor of the *RustHorn* approach is that the transformation only works for Safe Rust. While Unsafe Rust does not disable Rust's ownership semantics outright, allowing for the dereferencing of pointers makes simultaneous mutation and aliasing of data possible. So separation logic may still be required for Unsafe Rust.

3.3 Verification with Creusot

This section gives a practical overview of how to verify Rust software with Creusot. For a surface-level overview see §3.1.3.

3.3.1 Verification procedure

Given a Rust program that is already annotated for Creusot, how is one to proceed with proving the specification? To illustrate this, we will reuse the running example in Listing 3.3. First, it is important to understand that Creusot operates on complete Rust projects, that is, complete Rust libraries or applications. Typically Rust projects are built to be used with Cargo, the official building and package management tool. To work with this status quo, Creusot provides an extension for Cargo to conveniently use the tool for Rust projects.

The file structure of the finished project is as follows:

```
insert-creusot
├── Cargo.toml
├── prelude
│   ├── prelude.mlw
│   └── seq_ext.mlw
├── rust-toolchain
└── src
    └── lib.rs
```

The file `lib.rs` contains the code from Listing 3.3. But before compiling the project, it has to be set up appropriately. Remember that we used Creusot's contracts library in the running example. Cargo has to know where to fetch this external dependency from. To do this we add the following to the project's `Cargo.toml` file under the dependency section:

```toml
[dependencies]
creusot-contracts = { git = "https://github.com/xldenis/creusot" }
```

Typically, libraries are fetched from the central repository hosted at `crates.io`, but since Creusot is experimental software, it is only available from GitHub directly. Further, we need to provide a way of enabling this dependency's `contracts` feature. Conventionally, this is done by appending the following to `Cargo.toml`.

```toml
[features]
default = []
contracts = ["creusot-contracts/contracts"]
```

This defines two feature sets for our project: `default` and `contracts`, which in turn enables the `contracts` feature for `creusot-contracts`. This is because the `contracts` feature is necessary to generate the WhyML specification from the project with Creusot. The project would not regularly compile with this feature enabled. So the default feature set allows for regular compilation of the project, whereas `contracts` is used for verification. Lastly, the Rust compiler has to be set to a specific version. This version is the same one that Creusot was compiled with. Conventionally, this information is put into the `rust-toolchain` file in the project root where `channel` is set to the desired version:

```
[toolchain]
channel = "nightly-2023-01-15"
components = [ "rustfmt", "rustc-dev", "llvm-tools-preview" ]
```

When Cargo is invoked, this will download the appropriate version of the Rust compiler and adjacent tooling, if not already present. With this, the project is ready for the next step.

Generating the WhyML specification

When the project is set up appropriately, we invoke Creusot as follows:

```
cargo creusot -- --features=contracts
```

Note that we select the `contracts` feature for the previously explained reason. If the process exits successfully, one finds the generated specification in the `target/debug/` subdirectory with the file extension `.mlcfg`.

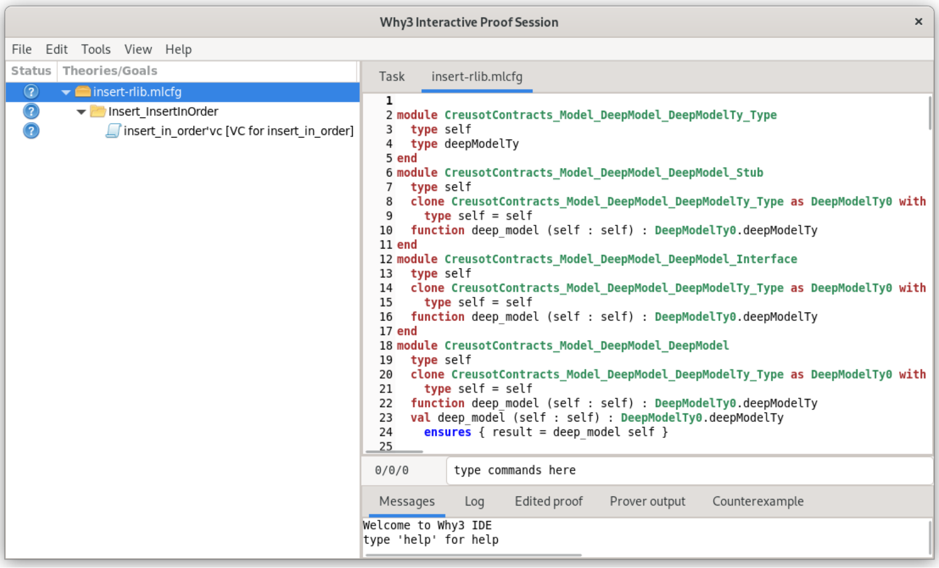
Loading the specification into Why3

The generated file is ready to be loaded into Why3 for proving. In addition to this file, we also have to load Creusot's WhyML prelude. This can be found in Creusot's main repository on GitHub. Now with both the generated file and the prelude dependencies, we can load them into Why3 like so:

```
why3 ide -L <path to prelude>/prelude target/debug/*.mlcfg
```

Proving in Why3

On opening the Why3 IDE, this is what we see:



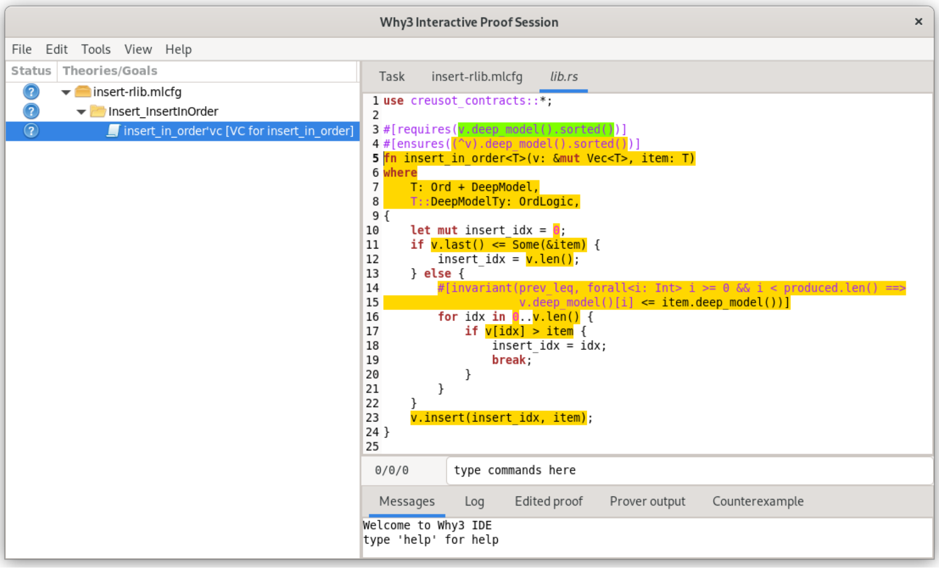
```

module CreusotContracts_Model_DeepModel_DeepModelTy_Type
  type self
  type deepModelTy
end
module CreusotContracts_Model_DeepModel_DeepModel_Stub
  type self
  clone CreusotContracts_Model_DeepModel_DeepModelTy_Type as DeepModelTy0 with
    type self = self
  function deep_model (self : self) : DeepModelTy0.deepModelTy
end
module CreusotContracts_Model_DeepModel_DeepModel_Interface
  type self
  clone CreusotContracts_Model_DeepModel_DeepModelTy_Type as DeepModelTy0 with
    type self = self
  function deep_model (self : self) : DeepModelTy0.deepModelTy
end
module CreusotContracts_Model_DeepModel_DeepModel
  type self
  clone CreusotContracts_Model_DeepModel_DeepModelTy_Type as DeepModelTy0 with
    type self = self
  function deep_model (self : self) : DeepModelTy0.deepModelTy
  val deep_model (self : self) : DeepModelTy0.deepModelTy
    ensures { result = deep_model self }
```

```
0/0/0          type commands here
```

```
Welcome to Why3 IDE
type 'help' for help
```

Most of the space is taken up by the contents of the loaded WhyML file. The sidebar on the left contains a root node for this file, and within, nodes for the individual proof goals Why3 extracted from the specification. Since our running example is simple, it contains only one goal node. By double-clicking on this node Why3 opens the original `lib.rs` file with coloured spans. Green represents statements that are assumed to be true, and yellow statements for which the validity is the goal to verify.



```
use creusot_contracts::*;

#[requires(v.deep_model().sorted())]
#[ensures((^v).deep_model().sorted())]
fn insert_in_order<T>(v: &mut Vec<T>, item: T)
where
    T: Ord + DeepModel,
    T::DeepModelTy: OrdLogic,
{
    let mut insert_idx = 0;
    if v.last() <= Some(&item) {
        insert_idx = v.len();
    } else {
        #[invariant(prev_leq, forall<i: Int> i >= 0 && i < produced.len() ==>
                    v.deep_model()[i] <= item.deep_model())]
        for idx in 0..v.len() {
            if v[idx] > item {
                insert_idx = idx;
                break;
            }
        }
    }
    v.insert(insert_idx, item);
}

```

```
0/0/0          type commands here
```

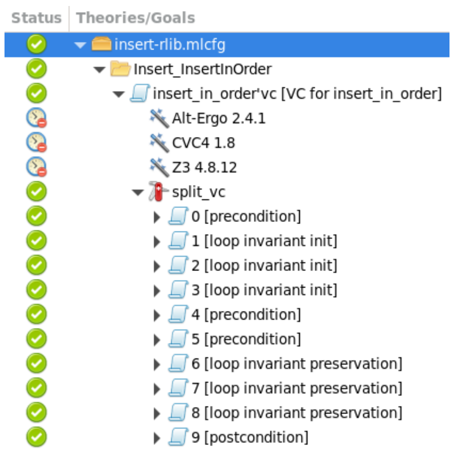
```
Welcome to Why3 IDE
type 'help' for help
```

Now we want to move on and prove this one goal node. Why3 offers four different levels of automating the proving process. The levels differ in the configured timeout for the external SMT solvers and if solvers produce inconclusive output, whether goal nodes should be split up into several smaller ones. A comparison of Why3's levels is found in Table 3.2. For our purposes, the highest level, 3, is appropriate. To select it we right-

Auto level	Timeout	Auto-splitting
0	short	
1	long	
2	short	✓
3	long	✓

Table 3.2: Overview of Why3's auto levels comparing prover timeout and if after the timeout is reached, the goal is split and proving is attempted on the subgoals. This behaviour is described in [Why22, p. 68].

click the goal node and select 'Auto level 3' from the context menu. Eventually, the root icon will turn green, indicating that the verification was a success[4]. If we expand the root node again, we can see that the SMT solvers failed to produce a conclusive result for the initial goal node. But since we selected 'Auto level 3', Why3 proceeded to automatically split the goal into ten sub-goals. And through repeated application of this procedure, the original goal was verified.



Status | Theories/Goals

insert-rlib.mlcfg
Insert_InsertInOrder
insert_in_order'vc [VC for insert_in_order]
Alt-Ergo 2.4.1
CVC4 1.8
Z3 4.8.12
split_vc
0 [precondition]
1 [loop invariant init]
2 [loop invariant init]
3 [loop invariant init]
4 [precondition]
5 [precondition]
6 [loop invariant preservation]
7 [loop invariant preservation]
8 [loop invariant preservation]
9 [postcondition]

3.3.2 Writing Creusot specifications

At the time of writing, Creusot does not have comprehensive or complete documentation of its features. This section documents features that will be important for understanding the presented work. It does not aim to be exhaustive. Information is assembled from [Sko22], [DJM22], and [DJ22].

[4]For the meaning of node icons see *Help>Legend* from the main menu.

Pearlite

Pearlite is Creusot's specification language. It adds logic constructs important for writing specifications but remains syntactically similar to Rust.

Logic constructs Creusot features familiar logic constructs, like implication and quantifiers. An implication like $a \implies b$ is written as `a ==> b`. Quantifiers are boolean statements for an instance `t` of some Rust type `T`. They are followed by another boolean statement (`stmt`) using `t`. The full statement is written as `quantifier<t: T> stmt`, where `quantifier` is replaced with `forall` and `exists` for universal and existential quantification respectively.

Final operator The final operator brings the core insight of *RustHorn* to Pearlite. By prefixing a variable that represents a mutable reference with the final operator `^`, it allows for reasoning about the referenced value at the time the reference dies. For example, this is how the running example in Listing 3.3 allows for specifying the value of `*v` on function exit since that is also the moment when `v` dies.

Special variables In specific contexts, Creusot exposes special variables to make the writing of specifications more convenient. The `result` variable is available when writing annotations for functions. It refers to the value the function returns. And the `produced` variable is available when writing invariant annotations for `for`-loops. In Rust a `for`-loop is always derived from an iterator, and `produced` holds the sequence of values that were already produced by the iterator. This is how the invariant for the running example in Listing 3.3 can refer to its length via `produced.len()` to restrict `i`. For details see [DJ22, §2.3].

Function purity

Regular Rust functions may not be called from within Pearlite. To call functions they have to be logically pure (i.e. free of side effects). Such functions are declared with the `#[logic]` annotation. Within, the `pearlite!`-macro may be used to write expressions in Pearlite. If such a logic function is boolean, it can also be declared as a `#[predicate]`. For example, instead of relying on Creusot's internal `sorted` predicated, we could define our own like in Listing 3.5. Note how we use the function `Seq::len()` within the `pearlite!`-block, which is only possible because it is a logic function, too.

Assertions

Standard Rust `assert!` and `debug_assert!` statements get converted to proof goals by Creusot. Further, Creusot offers the additional `proof_assert!`-macro. It functions

```
#[predicate]
fn is_sorted<T: OrdLogic>(seq: Seq<T>) {
    pearlite! {
        forall<m: Int, n: Int> m >= 0 && n >= 0 &&
            m < seq.len() && m < n ==>
            seq[m] <= seq[n]
    }
}
```

Listing 3.5: A custom Creusot predicate for checking if a `Seq<T>` is sorted.

similarly to `assert!` but explicitly exists for verification purposes. Statements inside are written in Pearlite. The assertion is not checked if the program is regularly compiled.

Annotations

- `#[requires()]` is a precondition for a function.
- `#[ensures()]` is a postcondition for a function.
- `#[maintains()]` is syntactic sugar that is converted into the same pre- and post-conditions for a function.
- `#[invariant(name, stmt)]` is an invariant that holds for each step of a loop (`for`, `while`, and `loop`). Each `invariant` annotation requires a `name`. Every loop requires such an annotation, even if `stmt` is a tautology.
- `#[trusted]` code is not checked by Creusot. Other annotations for the code are assumed to be valid. For example, this feature can be used to write specifications for unsafe Rust code, which Creusot does not otherwise support.

Traits

Creusot heavily utilises Rust's trait system to define properties of types.

Model The idea of models in Creusot is to have an abstraction from existing types that might contain additional complexity that is not required for verification. Creusot distinguishes between shallow and deep models. A shallow model applies the abstraction only to the type itself, whereas a deep model applies the abstraction recursively to contained types. Writing specifications using shallow models can be simpler because it doesn't require that the contained type provides a model as well. Deep models are for reasoning about the contained data. Listing 3.6 shows how models are defined as traits. Importantly, the model traits require a model type which the struct implementing the

```rust
pub trait ShallowModel {
    type ShallowModelTy;
    #[logic]
    fn shallow_model(self) -> Self::ShallowModelTy;
}

pub trait DeepModel {
    type DeepModelTy;
    #[logic]
    fn deep_model(self) -> Self::DeepModelTy;
}
```

Listing 3.6: The definition of the model traits in Creusot.

trait is then turned into when abstracted into a model. For example, for `Vec<T>`, the shallow model type is `Seq<T>` and the deep model type is `Seq<T::DeepModelTy>`. Here `Seq` is Creusot's abstraction for sequences in WhyML. Unlike `Vec`, `Seq` does not track allocation information, which is of little interest for verification. Pearlite's `@`-operator is syntactic sugar for calling `shallow_model()`.

OrdLogic The `OrdLogic` trait can be seen as an extension of Rust's standard `Ord` trait, which is used for defining total orders on types. `OrdLogic` gives additional properties of the ordering relation. It is reflexive, transitive, and antisymmetric. When doing comparisons of order with Pearlite, it is required that the compared type implements `OrdLogic`.

Trait extensions Some traits require Creusot-specific extensions of the same name but within Creusot's namespace. One example of this is the `Default` trait. Standard `Default` provides a default value for types implementing it. Creusot's extension also requires the definition of a predicate `Default::is_default()`. The specification for `Default::default()` uses it as follows:

```rust
#[ensures(result.is_default())]
fn default() -> Self;
```

Another such example is the `Iterator` trait. It requires that the `Iterator::completed()` predicate is implemented among other functions. It returns `true` if the iterator is exhausted and will not return any more items.

3.4 Runtime verification considerations

This section discusses in broad strokes how verifying the Rust LF runtime with Creusot was approached, including a summary of why Creusot was chosen from the previously presented tools.

3.4.1 Why Creusot

While there is no publicly available case study in verification work for existing medium-complexity Rust software, Creusot appears to be the best choice to make the attempt out of the tools presented in §3.1. Generally, Prusti appears to be too immature for the task, which leaves Creusot and Kani. Creusot's lack of support for Unsafe Rust and concurrency is certainly a limiting factor. Lack of support for Unsafe Rust less so, because the LF Rust runtime only has very few unsafe statements in its source. The lack of support for concurrency is a greater concern since deterministic concurrency is one of the unique properties of the Reactor Model, however, Kani does not support concurrency either. Another significant factor is that CreuSAT showcases Creusot's capability to work with Rust software projects of comparable size to the LF Rust runtime. Presumably, Kani is similarly battle-tested at AWS. The introductory publication on the tool does mention it being used to verify parts of Firecracker, AWS' virtual machine management tool, however, this work is not public. Additionally, Kani being a model checker, it does not scale well with custom properties one would wish to verify. Further, Kani can only reason about concrete programs, which is problematic for verifying a runtime in the abstract. Contrast this with Creusot, which is capable of abstract reasoning about Rust software through deductive verification. Not only does the verification of custom properties scale much better with this approach, it means that verified properties hold for every reactor program using the runtime, not just a singular instance.

3.4.2 Using Creusot

Due to some of Creusot's limitations, this work can only verify a subset of the Rust LF runtime. The lack of support for concurrency means that verification work can only be done on the runtime with the `parallel-runtime` feature disabled. But this also means that physical actions cannot be verified either, since those are scheduled on separate threads. Another limitation of Creusot is the lack of support for Unsafe Rust. This means that the few optimizations that use unsafe code will have to be reverted for verification.
Regarding the proofs themselves, when the generated WhyML is loaded into Why3, they should verify at most on auto level 3. This is because it makes the proofs reproducible from scratch, but without the complication of needing additional manual input.

3.4.3 Defining correctness properties

Before the verification work, the to-be-verified properties have to be laid out clearly. The ideal way of formulating the correctness properties would be similar to how the Creusot authors specified and verified iterators in Rust in [DJ22]. This would require defining a transition relation for the runtime state. While the published literature on Lingua Franca does define how the runtime should process events and triggers, there is no formal definition of the runtime state from which correctness properties could be derived. So this work will have to present its own encoding of the state of a reactor program, with which one can formulate properties as logical expressions. These would then easily translate into similar expressions in Pearlite.

4 Properties of a correct runtime

This section presents several correctness criteria for the LF Rust runtime. Some of these properties will be general and applicable to other implementations and others will be highly specific to the Rust implementation. Every property is accompanied by an explanation of why the property is desirable and important for the runtime's correctness.

4.1 Termination

Termination of reactor programs is not necessarily a desired property. For example, a reactor program that acts as a server should, ideally, be able to run for indefinite amounts of time. This would mean that within the runtime the main event loop never terminates. However, that individual iterations of the loop terminate is a desirable property and important for the correctness of the runtime. If an individual iteration of the main event loop does not terminate, it would mean, that events are not processed by the runtime. And even if the loop is implemented perfectly, it could still be that the body of an invoked reaction does not terminate. These considerations lead to the first correctness property: *assuming that the invoked reaction bodies within the main event loop terminate, the individual iteration of the loop terminates also.*

4.2 Crash freedom

Crashes are a generally undesired occurrence for programming platforms like Lingua Franca. The programmer, of course, may still introduce crashes into the program, but the foundations like the runtime should not crash on their own. Since programmers generally interact with the programming environment through abstractions, only a few will have a holistic understanding of the full functionality of their program. Thus crashes not caused by the programmer's code would be unanticipated. This could at worst lead to security problems where the deliberate crashing of the program could be exploited to facilitate a denial of service attack. Rust's memory safety already prevents certain classes of bugs that are a common cause of crashes in programming languages like C, but certain APIs may still deliberately trigger a crash on certain input and Rust's safety guarantees are significantly relaxed in Unsafe Rust. This leads to the second correctness property: *the runtime does not cause crashes.*

4.3 Correctness of VecMap

`VecMap` is the only custom generic data type implementation within the runtime's codebase. It is a simple map implementation that stores key-value pairs and allows data to be retrieved by the associated key. The data is tightly packed in memory by using Rust's standard `Vec` type underneath. To minimise key lookup time, the key-value pairs are retained in sorted order by their key. This data type is used in multiple places across the code base:

- during initialisation when building the dependency graph,
- when storing the values scheduled for a specific action and tag,
- when storing the triggered reactions for a tag in the event queue (grouped by level).

So the correct implementation of this type is vital for the correct functioning of the runtime. And while the runtime does depend on similar other data types as external dependencies like `index_vec` and `smallvec`, these are very popular Rust project dependencies[1]. While popularity does not ensure correct implementation, it is reasonable to be much more confident about the correctness of their implementation, with many Rust projects choosing to use them.
Laying out concrete correctness criteria for `VecMap` is hard, because the data structure has an extensive API. Ideally, it is possible to prove that each of `VecMap`'s functions satisfies the contract described in their documentation. But there is one invariant they all rely on: key-value pairs are stored in sorted order of their keys. This is because item retrieval uses binary search that assumes items are correctly ordered. If that assumption is not true, binary search will not work correctly. This results in the third correctness property: *as long as an instance of `VecMap` exists, its stored items are sorted in key order.*

4.4 Formal properties

The previously expressed properties are hard to express formally. This section presents properties expressed in first-order logic. These properties are defined over the execution of the main event loop (described in §2.4.2). It makes up the biggest amount of code that is executed consecutively, without interruption from reaction bodies or code from the code generator. Thus it is an attractive target over which to define runtime correctness properties.

[1]As of the 30th of January 2023, `index_vec` has 57 thousand downloads on `crates.io` and `smallvec` has 111 million.

4.4.1 Reactor program state

To be able to define formal properties over the execution of reactor programs, a formalisation of the state of reactor programs is needed. This state is made up of the runtime's state, plus the description of the reactor program.

Definition 4.4.1. A reactor program's state is a tuple $\xi = (R, R_0, N, \prec, g, \mathcal{V}, Q_E, Q_R, T_R)$ where

- R is the set of reactors of the program,
- $R_0 \in R$ is the top-level reactor,
- N is the set of reactions of the program,
- $\prec$ is the partial order of reactions in N,
- $g \in \mathbb{G}$ is the current logical time,
- $\mathcal{V} \colon \Sigma \longrightarrow V$ is the valuation mapping,
- Q_E is the event queue (set),
- Q_R is the reaction queue (set),
- $T_R \in \mathbb{N} \times \mathbb{G} \times N$ is the executed reactions trace.

Definition 4.4.2. The set Ξ is the set of all reactor program states.

The static information about the program found in R, R_0, N, and $\prec$ is highly redundant. Theoretically, these sets could all be derived from just R_0. However, making the information redundantly available like this simplifies the writing of properties. $\mathcal{V}$ maps the identifier of a reactor component, like input/output or state variable to its current value. While Q_E and Q_R are called queues, this is to describe their role in this representation and the wider literature, not their functionality. They are sets in this abstraction. Finally, T_R is a construction that is absent in the LF runtime, but important for the formal definition of correctness properties. It contains every executed reaction together with the logical time it was executed at. But since logically simultaneously observed reactions are in practice processed sequentially, T_R additionally stores a constantly increasing index for each reaction.

The last thing needed for describing formal properties is a transition relation from one reactor state to the other.

Definition 4.4.3. $\Downarrow \colon \Xi \longrightarrow \Xi$ is the reaction execution relation, where the execution of a reaction according to RM semantics leads to the resulting reactor program state.

Since reactions are the atomic computational components of the RM, using their execution as the basis for defining distinct reactor program states makes sense. A proper formal definition of $\Downarrow$ is beyond the scope of this work.

Definition 4.4.4. $\Downarrow^+\colon \Xi \longrightarrow \Xi$ is the transitive closure of $\Downarrow$, i.e.

$$\forall \xi, \xi' \in \Xi\colon \xi \Downarrow \xi' \implies \xi \Downarrow^+ \xi'$$

and

$$\forall \xi, \xi', \xi'' \in \Xi\colon \xi \Downarrow \xi' \wedge \xi' \Downarrow^+ \xi'' \implies \xi \Downarrow^+ \xi''.$$

With these definitions, it is now possible to define correctness criteria.

4.4.2 Constant program specification

The specification of a reactor program stays constant during execution. This may not be true once mutations are implemented, but since LF does not implement them, this is of no concern. The specification is as follows:

$$\forall \xi, \xi' \in \Xi\colon \xi \Downarrow^+ \xi' \implies R(\xi) = R(\xi') \wedge R_0(\xi) = R_0(\xi') \wedge N(\xi) = N(\xi') \wedge \prec(\xi) = \prec(\xi')$$

4.4.3 Linear time

The RM semantics require that time moves forward. This leads to the following property:

$$\forall \xi, \xi' \in \Xi\colon \xi \Downarrow^+ \xi' \implies g(\xi) \le g(\xi')$$

4.4.4 Reaction precedence derived correctly

LF's promise of determinism relies on the reaction precedence relation $\prec$ being derived correctly from the program specification. For a definition of $\prec$ see §2.3.5. This definition can be translated into the notation here, turning it into a correctness property that verifies that $\prec$ was derived correctly:

$$\begin{aligned} &\forall \xi \in \Xi . \forall n, n' \in N(\xi)\colon n \prec n' \implies \\ &\qquad (\exists d \in D(n') : d \in D^\vee(n)) \vee \\ &\qquad (\exists r \in R(\xi) : n \in N(r) \wedge n' \in N(r) \wedge P(r)(n) < P(r)(n')) \end{aligned}$$

4.4.5 Event processing

This section groups formal correctness properties related to processing events in the main event loop. It is important to note here that in the RM, the value associated with an event is overwritten when the same action is scheduled for the same tag.

Reaction execution precedence

In the same way that it is important for the reaction precedence relation $\prec$ to be derived correctly, it is equally important that reactions are actually executed according to it:

$$\begin{aligned}\forall \xi, \xi' \in \Xi . \forall e, e' \in Q_E(\xi) . \forall n, n' \in N(\xi) . \exists j \in \mathbb{N} \colon \xi \Downarrow^+ \xi' \wedge \\ g(e) = g(e') \wedge t(e) \in T(n) \wedge t(e') \in T(n') \wedge \\ n \prec n' \wedge (j, g(e'), n') \in T_R(\xi') \implies \\ \exists i \in \mathbb{N} \colon j > i \wedge (i, g(e), n) \in T_R(\xi')\end{aligned}$$

Events stay on Q_E until triggers are processed

An event stays on the event queue (Q_E) as long as the reactions triggered by the event aren't executed:

$$\begin{aligned}\forall \xi, \xi' \in \Xi . \forall e \in Q_E(\xi) . \forall n \in N(\xi) . \exists i \in \mathbb{N} \colon \xi \Downarrow^+ \xi' \wedge \\ (i, g(e), n) \in T_R(\xi') \wedge t(e) \in T(n) \implies \\ g(e) \leq g(\xi') \wedge \neg \exists e' \in Q_E(\xi') \colon g(e) = g(e') \wedge t(e) = t(e')\end{aligned}$$

Again, it is important to remember that for scheduled events the associated value may change. This is why the final expression does not simply read $e \notin Q_E(\xi')$.

Events are only removed from Q_E after their designated time

This last property ensures that events stay on the queue until their assigned tag is reached. Otherwise, with the previous properties, the execution could happen earlier, which, of course, is not desired. The property is as follows:

$$\begin{aligned}\forall \xi, \xi' \in \Xi . \forall e \in Q_E(\xi) \colon \xi \Downarrow^+ \xi' \wedge g(\xi') < g(e) \implies \\ \exists e' \in Q_E(\xi') \colon g(e) = g(e') \wedge t(e) = t(e')\end{aligned}$$

5 Verifying the LF Rust runtime

This section details the results of attempting to verify Lingua Franca's Rust runtime in reference to the correctness criteria laid out in §4. The first part details the attempt at applying Creusot to the whole runtime and the second part goes over the verification of the `vecmap` module within the runtime. Both parts outline their general approach and list the problems that were encountered, and how the problem was overcome if a solution was found.

5.1 Verifying the whole runtime

Unfortunately, verifying a single module of the runtime with Creusot is not feasible. This is because the function preconditions have to be valid when used, and often it is the case that a function is defined in one module and then used in another. This is why the initial verification attempt covers the entire runtime.
Throughout this section, the `time` module is used as an illustrative example for the changes that had to be made.

5.1.1 Goals

Covering the entire runtime with Creusot means that the initial verification goal should not be one of the specific and detailed correctness criteria from §4.4. This is because the function preconditions necessary for these properties would also apply to and have to be specified for their call sites, meaning that they balloon into further pre- and postconditions and so forth. Among the remaining correctness properties, the logical starting point would be proving crash freedom as detailed in §4.2, because it requires covering the entire code base, while not being too specific. Since Creusot checks for the absence of crashes by default, the derived annotations would be the minimum required to progress to any of the other correctness properties. These annotations would involve, for example, requiring that integers do not overflow.
Additionally, this verification work should not alter the outward API which the LF code generator relies on, because this avoids having to adjust the code generator for changes in the API, or the rewriting of existing reactor programs if the API of `ReactionCtx` was touched.

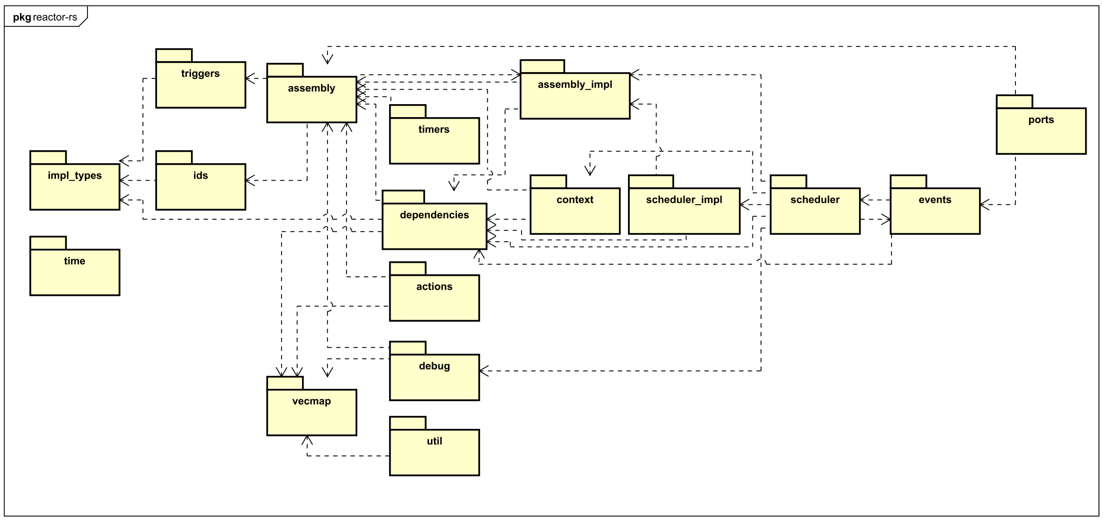


Figure 5.1: A graph illustrating the dependencies of the Rust LF runtime modules. The modules with the least coupling are leftmost with no outgoing dependency relations. This means that only the modules `impl_types`, `time`, and `vecmap` have no coupling.

5.1.2 Approach

Verifying an existing large code base with Creusot is challenging because the tool automatically covers all code in the compilation process. And doing the verification module-by-module is not possible, since the different modules that make up the runtime are strongly dependent on each other. See Figure 5.1 for an illustration of the runtime's module dependency graph. With the exception of `impl_types`, `time`, and `vecmap`, every module has some degree of coupling that makes the module-by-module approach impossible. Further, `impl_types` and `time` are very small modules that mostly only contain simple type declarations.

The strategy that was decided upon for this work was to comment out parts of the code that would trigger this dependency ripple effect, and then gradually activate the commented code again until the whole runtime is covered. During this process, Creusot-specific traits like `ShallowModel`/`DeepModel` are implemented as necessary. To not disrupt the ongoing development and usage of the runtime library, it was forked into a separate repository[1]. The Creusot version used for this section is identified by Git commit hash `edb39db`.

[1]The forked runtime with the changes made throughout this work is located at `https://github.com/jhaye/reactor-rs`.

5.1.3 Challenges and solutions

During the verification attempt, several issues were encountered that this section details. Most of these problems relate to lacking support for Rust language features in Creusot. For some of these issues, a solution was found, which will also be explained.

No support for concurrency

Creusot's lack of support for concurrency was well understood before this verification attempt started. The Rust runtime uses the external dependency `rayon` to execute reactions in parallel. During the initialisation of the runtime, a pool of worker threads is created and the loop executing reactions level-by-level is parallelised. The effects are then merged and the loop starts over. Executing reactions of the same level in parallel like this works since they, by definition of the level, are independent of each other. This way of parallelising the work is relatively simple, which also made it easy to remove. There is no reason to keep functionality that cannot be verified. This also greatly simplifies the dependency tree of the runtime. As seen in Figure 2.7, `rayon` and its respective dependencies make up a significant part of the runtime's dependencies.

No support for Unsafe Rust

The lacking support for Unsafe Rust in Creusot was also anticipated. Since the runtime was constructed knowing of the issues that using Unsafe Rust could introduce, the runtime contains relatively few `unsafe` blocks. This is why the same approach as for concurrency was taken. All `unsafe` blocks within the runtime were removed and replaced with previous, less efficient, but verifiable implementations. Overall, 20 uses of `unsafe` were removed.

No support for `const`

The Creusot version used in this verification attempt does not support constant expressions[2]. This means that both `const` functions, as well as variables crash Creusot. However, this is only a minor issue, because constants can be rewritten to not be constants. For example, the `time` module defines a type `MicroStep` with the constant `ZERO`:

```rust
pub struct MicroStep(u32);

impl MicroStep {
    pub const ZERO: MicroStep = MicroStep(0);
}
```

[2]This problem is partially documented in Creusot's GitHub issue #358.

This type is the runtime's implementation of the micro-step index of a tag—a simple 32-bit integer essentially. But since Creusot does not support constants, `MicroStep::ZERO` cannot be used. The solution is to comment out the constant's definition and to substitute its usage with `MicroStep(0)`.

No support for `impl`/`dyn` traits

While the lack of support for constants was a minor obstacle, the lack of support for `impl`/`dyn` traits is not[3]. This Rust feature is for restricting types of function arguments and return values to specific traits that they implement. While `impl` traits are just syntactic sugar that is resolved at compile time, `dyn` traits provide polymorphism through dynamic dispatch. For example, imagine we had a function that takes an argument of type `&mut dyn Write` and in its body called the `write` function this trait provides, and further, that this function is called separately with the concrete types `File` and `TcpStream`. Both types implement `Write`, but because the argument is `dyn`, the decision of which specific instance of `write` to call for these concrete types is made at runtime.
The problem is that these language features are used both at the core of the runtime as well as the Rust standard library. For example, the `Debug` and `Display` traits are Rust's idiomatic way of string formatting the instances of Rust structs. Both traits are structured very similarly, providing a `fmt` function that takes a `Formatter` as an argument which looks like this:

```rust
pub struct Formatter<'a> {
    flags: u32,
    fill: char,
    align: rt::v1::Alignment,
    width: Option<usize>,
    precision: Option<usize>,
    buf: &'a mut (dyn Write + 'a),
}
```

The `buf` member has the `dyn Write` trait, which means it will cause Creusot to crash, and consequently `Debug` and `Display` do the same. This means that any implementation of these traits has to be commented out and any code using it (e.g. for logging or printing to the console) also. Further, the runtime uses `dyn` traits for reaction execution, meaning that this crucial part of the runtime cannot be verified either.
Unfortunately, there is no alternative to this language construct compared to, for example, constants which can be replaced by non-constants. The feature is deeply embedded in Rust's standard library and code using it would have to be greatly reconstructed to work without it. And marking the code with `#[trusted]` does not work either. The code that is marked as such is still processed by Creusot, triggering the crash.

[3]This problem is reported in Creusot's GitHub issues #539 and #610.

Insufficient coverage of standard library

Any invoked function that is checked with Creusot needs a contract. For functions and types from the standard library, Creusot comes with `creusot-contracts`, which provides contracts for them. However, the overall coverage of the standard library is small. This prompted the writing and contributing of new contracts since the Rust LF runtime heavily utilises the standard library.
To give an example, to measure time, the runtime uses the standard-library-provided types `Instant` and `Duration`. The runtime records a startup instant t_0, and whenever it needs to measure the duration since startup, it creates a new `Instant` and with this and t_0 a `Duration` representing this value can be generated. But neither type had any defined contracts when starting the work.

The `Duration` type stores two values: the number of elapsed seconds and the number of elapsed sub-second nanoseconds. The first thing to implement is the model traits. `Duration` records seconds and nanoseconds, so it is easiest to convert it into overall nanoseconds, making the model type `Int`, WhyML's generic unrestricted whole integer type. The problem is that the internal value for a `Duration` is not accessible, so the state of it used in the contracts has to be defined axiomatically. For the model functions this means that we cannot derive what the internal value is, but merely what a valid value for `Duration` is. While the seconds part of `Duration` is a 64-bit unsigned integer, `u64`, which is fully used, the sub-seconds part must be lower than one second, meaning its valid range is from 0 to 999,999,999 nanoseconds. Furthermore, durations are definitionally positive. The model trait implementations for `Duration` with these restrictions can be seen in Listing 5.1. Since `Duration` does not contain any other generic type, both `ShallowModel` and `DeepModel` ought to return the same value. The `secs_to_nanos` function is a simple unit conversion function. The `absurd` statement in the function body is a WhyML construct, representing code that shouldn't be reached, which is the case since the functions are marked as `#[trusted]`. The utility in this open definition is that when through one of `Duration`'s functions the internal value is set, we can define what the value of the model is. The excerpt of the annotated implementation of `Duration` in Listing 5.2 demonstrates this. For `new` it is important to require that the seconds part of a `Duration` does not overflow since the `nanos` parameter is converted and added to it if it is greater or equal to a second. Arithmetic on `Duration` is also defined and used within the runtime. Similar precautions to prevent integer overflows have to be made there, as can be seen in Listing 5.3. Writing contracts for the `Instant` type is very similar.

```
impl ShallowModel for Duration {
    type ShallowModelTy = Int;
    #[logic]
    #[trusted]
    #[ensures(result >= 0 &&
              result <= secs_to_nanos(@u64::MAX) + 999_999_999)]
    fn shallow_model(self) -> Self::ShallowModelTy {
        pearlite! { absurd }
    }
}

impl DeepModel for Duration {
    type DeepModelTy = Int;
    #[logic]
    #[trusted]
    #[ensures(result == self.shallow_model())]
    fn deep_model(self) -> Self::DeepModelTy {
        pearlite! { absurd }
    }
}
```

Listing 5.1: The implementation of the model traits for `Duration`.

```
 impl Duration {
    #[requires(@secs + nanos_to_secs(@nanos) <= @u64::MAX)]
    #[ensures(@result == secs_to_nanos(@secs) + @nanos )]
    fn new(secs: u64, nanos: u32) -> Duration;

    #[ensures(@result == secs_to_nanos(@secs))]
    fn from_secs(secs: u64) -> Duration;

    #[ensures(@self == 0 ==> result == true)]
    #[ensures(@self != 0 ==> result == false)]
    fn is_zero(&self) -> bool;

    #[ensures(@result == @self)]
    #[ensures(@result <= secs_to_nanos(@u64::MAX) + 999_999_999)]
    fn as_nanos(&self) -> u128;
}
```

Listing 5.2: A selection of contracts for `Duration`.

```rust
impl Add<Duration> for Duration {
    #[requires(@self + @rhs <= secs_to_nanos(@u64::MAX) + 999_999_999)]
    #[ensures(@self + @rhs == @result)]
    fn add(self, rhs: Duration) -> Duration;
}
```

Listing 5.3: The contract for the `Add` trait for `Duration`.

Contributed specifications for Creusot Standard library specifications created for this verification work were contributed back to the Creusot project. Overall, for this work the following specifications were added to `creusot-contracts`:

- specifications for `std::time::Duration`,
- specifications for `std::time::Instant`,
- specifications for `std::collections::VecDeque`,
- specifications for `std::result::Result`,
- the implementation of `DeepModel` and `OrdLogic` for `std::cmp::Reverse`.

No support for many trait derivations

For many traits, Rust has support for automatically deriving them. For example, for the `MicroStep` type it looks like this:

```rust
#[derive(Debug, Eq, PartialEq, Ord, PartialOrd, Copy, Clone, Hash)]
pub struct MicroStep(u32);
```

Since `MicroStep` is just the primitive `u32` underneath, deriving `Eq` (equality), `PartialOrd` (partial order) or `Ord` (total order) is possible. With the exception of `Debug` for aforementioned reasons, these traits can all be implemented with Creusot as well, but only few can be auto-derived[4]. This is because the automatic derivation also has to provide Creusot-specific information, like annotations and trait implementations. For this example, Creusot supports auto-deriving `Clone` and `PartialEq`, and `Copy` and `Eq` can be auto-derived as well, since they directly rely on those traits. But Creusot does not offer this functionality for `Ord` or `PartialOrd`—they have to be implemented manually. The manual implementation of these traits is depicted in Listing 5.4. Notice how the manually implemented traits are annotated. Also, notice how the specific auto-derived traits have to be imported explicitly from `creusot_contracts`.

[4]This is reported in Creusot's GitHub issue #609.

```rust
use creusot_contracts::{Clone, PartialEq, *};

#[derive(Eq, PartialEq, Copy, Clone)]
pub struct MicroStep(u32);

impl Ord for MicroStep {
    #[ensures(result == self.deep_model().cmp_log(other.deep_model()))]
    fn cmp(&self, other: &Self) -> Ordering {
        self.0.cmp(&other.0)
    }
}

impl PartialOrd for MicroStep {
    #[ensures(result == Some(self.deep_model()
                                 .cmp_log(other.deep_model())))]
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
```

Listing 5.4: Manual trait implementations for `MicroStep`.

No special support for working with dependencies

Creusot heavily relies on Rust's trait system, providing and requiring the implementation of several custom traits specific to Creusot. This however turns problematic with the usage of external dependencies. When external types are used in a Rust program and we wish to verify it with Creusot, then those types will also be a part of the annotations we write. At that point, the external type's implementation of Creusot-specific traits like `DeepModel` or `OrdLogic` is desirable, if not necessary. But this is not possible within the program using the external dependency, because of orphan rules. Put briefly, orphan rules require that if one wants to implement some trait `A` for some type `T`, at least one of them has to be local (i.e. defined in the same project the implementation happens) [Rusa]. The Rust Book further explains the necessity for orphan rules: 'This rule ensures that other people's code can't break your code and vice versa. Without the rule, two crates could implement the same trait for the same type, and Rust wouldn't know which implementation to use' [KN19, p. 184].
This makes the implementation of Creusot-specific traits for external types impossible because they are not local. That means that the only way of implementing Creusot-specific traits for external dependencies requires the forking of that dependency and implementing the traits directly.

5.1.4 Conclusion

The verification of the Rust LF runtime with the Creusot version used in this section failed. The core obstacle is the lack of support for `impl/dyn` traits. Every other problem listed in this section can be worked around, even if with significant additional work in some cases.
An improved version of Creusot that addressed all the limitations listed here, including broad or even full coverage of the standard library with contracts would significantly reduce the effort required for verifying existing Rust software like this runtime. But even for mature tooling, the effort required should not be underestimated. The work presented here does not allow for an estimate of how much time a full verification of the runtime in its current state would take.

5.2 Verifying VecMap

With the verification of the whole runtime out of the picture, this work's focus shifted to verifying individual parts of the runtime. As explained in the previous section, this is very difficult, because of the many module interdependencies within the runtime. But one module stands out with no coupling at all: `vecmap`. This section covers the verification process for this module.

5.2.1 Goals

Since Creusot operates on the crate level of a Rust project, not the module level, `vecmap` cannot be verified as is. But because there is no coupling, the module can very easily be turned into a stand-alone crate. This requires very little refactoring work within the runtime, mostly changing `use` statements from crate-internal to external ones. As laid out in §4.3, the core property that should be verified is that items stored in a `VecMap` stay sorted by their key throughout. Roughly this requires defining a predicate that checks if a `VecMap` is sorted and then requiring it for every function call of the type. Further, if the function can mutate the internal state (`&mut self` in function signature), the sorted property is ensured via post-condition. And simultaneously this verification work should pay attention to what individual functions do and verify their functionality, not just that internal storage stays sorted.
Additionally, `vecmap` is relatively small compared to the full runtime and its functionality is nicely abstracted with its API that the rest of the runtime uses. This means that this verification work can be done on the proper runtime, instead of forking it like in the previous section. But this comes with concerns over which Rust version to use. Creusot requires Nightly Rust, however, previous work spent effort on making the Rust LF runtime work with Stable Rust. This is because when one dependency requires Nightly Rust, the entire project has to be compiled with Nightly Rust. Ideally, the code used for verification of `vecmap` and the code used within the runtime would be one and the

```rust
pub struct VecMap<K, V>
where
    K: Eq + Ord,
{
    v: Vec<(K, V)>,
}
```

Listing 5.5: The definition of `VecMap`.

same, however in order to ensure that the runtime remains usable with Stable Rust that would not work. To satisfy this constraint, there are two versions of `vecmap`, identical in function, one for use and one for verification.

5.2.2 Defining the sorted property

The `VecMap` type's definition is in Listing 5.5. Key (`K`) and value (`V`) are defined as type parameters. And key-value pairs are stored in a regular `Vec` as tuples.
The first thing that is necessary for defining a predicate that checks if a `VecMap` is sorted, is requiring that its keys can be compared and ordered. Regular Rust does this via the `Ord` trait, as `VecMap` requires throughout its definition. But for Creusot this is not enough—they have to implement its specific `OrdLogic` trait. One way of resolving this is to require in the implementation, that provides the sorted predicate, that `K` implements `DeepModel` and further that the model type implements `OrdLogic`. The last important thing to remember for this property is that keys within a `VecMap` are unique. Since items are inserted into the map in ascending order of their key, this means for two items in the map that are not equal, the key of the one with the lower index in the internal map is strictly lesser than the other, not lesser or equal. The sorted predicate is defined in Listing 5.6.

5.2.3 Challenges and solutions

This section, similarly to the previous section, details the encountered challenges and solutions in this verification work.

Unsupported features

The Creusot version used for this verification is the same as for the verification attempt for the whole runtime. This means that the limitations discussed in the previous section also apply here. Luckily, `VecMap` does not use `impl` or `dyn` traits, so verification is actually possible. Technically this is not fully accurate, because like most data types, `VecMap` implements `Debug` and the module also contains a `Display` implementation. But

```rust
impl<K, V> VecMap<K, V>
where
    K: Eq + Ord + DeepModel,
    K::DeepModelTy: OrdLogic,
{
    #[logic]
    #[trusted]
    #[ensures(result.len() == (@self.v).len() &&
              forall<i: Int> i >= 0 && i < (@self.v).len() ==>
              result[i] == (@self.v)[i].0.deep_model())]
    fn key_seq(self) -> Seq<K::DeepModelTy> {
        pearlite! { absurd }
    }

    #[predicate]
    fn is_sorted(self) -> bool {
        pearlite! {
            forall<m: Int, n: Int> m >= 0 && n >= 0 &&
                m < (@self.v).len() && n < (@self.v).len() && m < n ==>
                self.key_seq()[m] < self.key_seq()[n]
        }
    }
}
```

Listing 5.6: The definition of the predicate to check if a `VecMap` is sorted, as well as a logic function that extracts the sequence of keys in a `VecMap`.

```rust
#[cfg(not(feature = "contracts"))]
impl<K: Ord + Eq + Debug, V: Debug> Debug for VecMap<K, V> {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        self.v.iter().collect::<Vec<&(K, V)>>().fmt(f)
    }
}
```

Listing 5.7: Using conditional compilation to not implement `Debug` when verifying.

since these are used for printing, they do not contribute to `VecMap`'s functionality that is being verified here. However, these implementations cannot simply be removed, since the runtime depends on them. A useful feature for this case is conditional compilation. Rust allows conditionally compiling designated code blocks if certain feature flags are set or not. As is convention for other Creusot projects, compiling for verification requires that the `contracts` feature is enabled. This can be used to not compile the code implementing `Debug` for `VecMap`, like in Listing 5.7. Another function that was chosen to be conditionally disabled like this is `VecMap::iter`. The function simply forwards the call to the `Vec`'s iterator function. And since this function is not used within the implementation itself and is trivially correct, assuming the standard library one is, the work of writing a specification for it was deemed not to be worth the effort.

Closures

Closure expressions are a language feature in Rust known under multiple names in other programming languages, like lambda functions/expressions or anonymous functions. They are functions that are not associated with an identifier and can capture parts of their environment. They often serve as arguments for higher-order functions. Creusot does have support for closures, but with the version used, writing specifications for higher-order functions is very involved [DJ22]. So much so that rewriting code using closures to not use them is much easier. The function `VecMap::max_key` illustrates the simplicity for verification when closures are eliminated. Originally, the function read:

```rust
pub fn max_key(&self) -> Option<&K> {
    self.v.last().map(|e| &e.0)
}
```

Since the underlying `Vec` is continually sorted in ascending order, the last element has the maximum key in the map. But a `Vec` can be empty, which is why `Vec::last`'s return type is `Option<T>`. `Option` is an enum that either is `None`, or `Some(t)` where `t` is some instance of `T`. Now, the internal `Vec` of the map does not store keys on its own, but instead, stores key-value pairs as tuples. So the last key-value pair is extracted with `Vec::last` if it exists. The ideal way of just extracting the key of the pair, while retaining the information about an empty vector is with `Option::map`. It takes a closure as an

```
#[requires(self.is_sorted())]
#[ensures(result == None ==> (@self.v).len() == 0)]
#[ensures(forall<k: &K> result == Some(k) ==>
          forall<i: Int> i >= 0 && i < (@self.v).len() ==>
          self.key_seq()[i] <= k.deep_model()
)]
pub fn max_key(&self) -> Option<&K> {
    match self.v.last() {
        Some(e) => Some(&e.0),
        None => None,
    }
}
```

Listing 5.8: The `max_key` function without the closure.

argument which is applied to the encapsulated value `Option`, if it is `Some(j)`, otherwise `None` stays untouched. For the purpose here the closure takes the key-value tuple and extracts the first element—the key.
How should the closure be eliminated? One simple way is rewriting `Option::map`'s behaviour with a `match` expression like in Listing 5.8. This code already contains the necessary annotations for Creusot. The behaviour of the code is unchanged. And with this, it is possible to verify the function without writing a specification for `Option::map`, which would undoubtedly be a much more complicated undertaking. But the removal of closures was not always so simple. The `VecMap::find_k` function is used whenever elements in the map are searched. Originally, the function looked like this:

```
fn find_k(&self, key: &K) -> Result<usize, usize> {
    self.v.binary_search_by_key(&key, |(k, _)| k)
}
```

The function simply forwards the result of `binary_search_by_key`, which returns `Ok(i)`, if `key` is in the map at index `i`, and `Err(j)`, if `key` is not in the map, but `j` is the index where an item with `key` could be inserted to maintain sorted order. The important part is the second argument which is the key-extraction function provided as a closure. The extraction function is applied to every element in the search and then compared to the supplied `key`. Replacing this closure requires the manual inlining of the binary-search function and applying the key extraction where usually the closure would be called. The resulting code, plus annotations are in Listing 5.9. The key extraction happens on line 31. Because of the many integer variables in the function, the `while` loop that forms the heart of the search function is annotated with many invariants restricting their values. And while, again, this might seem quite complicated, writing a specification for `binary_search_by_key` was attempted by Creusot lead author Xavier Denis and deemed too complicated, compared to the inlining described above.

```
#[requires(self.is_sorted())]
#[ensures(match result {
    Ok(_) => self.key_seq().contains(key.deep_model()),
    Err(_) => !self.key_seq().contains(key.deep_model()),
})]
#[ensures(forall<i: usize> result == Ok(i) ==> self.key_seq()[@i] == key.deep_model())]
#[ensures(forall<i: usize, j: Int> result == Err(i) ==> j >= @i && j < (@self.v).len() ==>
          self.key_seq()[j] > key.deep_model())]
#[ensures(forall<i: usize, j: Int> result == Err(i) && j >= 0 && j < @i ==>
          self.key_seq()[j] < key.deep_model())]
#[ensures(match result {
    Ok(idx) => @idx < (@self.v).len(),
    Err(idx) => @idx <= (@self.v).len(),
})]
fn find_k(&self, key: &K) -> Result<usize, usize> {
    let mut size = self.v.len();
    let mut left = 0;
    let mut right = size;
    let mut mid;
    #[invariant(size_bounds, @size >= 0 && @size <= (@self.v).len())]
    #[invariant(left_bounds, @left >= 0 && @left <= (@self.v).len())]
    #[invariant(right_bounds, @right >= 0 && @right <= (@self.v).len())]
    #[invariant(mid_bounds, @left < @right ==> (@left + (@size / 2)) < (@self.v).len())]
    #[invariant(right_gt_mid, @left < @right ==> @right > (@left + (@size / 2)))]
    #[invariant(right_geq_key, forall<i: Int> i >= @right && i < (@self.v).len() ==>
                self.key_seq()[i] > key.deep_model())]
    #[invariant(left_lt_key, forall<i: Int> i >= 0 && i < @left ==>
                self.key_seq()[i] < key.deep_model())]
    while left < right {
        mid = left + size / 2;
        let cmp = self.v[mid].0.cmp(key);
        match cmp {
            Ordering::Less => left = mid + 1,
            Ordering::Greater => right = mid,
            Ordering::Equal => return Ok(mid),
        }
        size = right - left;
    }
    Err(left)
}
```

Listing 5.9: The fully annotated version of `find_k`. This verifies in Why3.

```rust
impl<K, V> VecMap<K, V>
where
    K: Eq + Ord + DeepModel,
    K::DeepModelTy: OrdLogic,
{
```

Listing 5.10: The start of `VecMap`'s `impl` block.

```rust
#[cfg(not(feature = "contracts"))]
trait Key: Ord + Eq {}

#[cfg(feature = "contracts")]
trait Key: Ord + Eq + DeepModel
where
    Self::DeepModelTy: OrdLogic,
{}

impl<K, V> VecMap<K: Key, V> {
```

Listing 5.11: Sketching an idea for using supertraits to keep verification and actually used code in one file.

Same-file restriction

The goal for this verification was for the code ultimately used in the runtime and used for verification to be one and the same. This means the code would come from the same source file. The rationale is that translation of the verified version of `VecMap` into an intermediate form to be used in the runtime could introduce errors. Because the runtime requires Stable Rust, all usages of Creusot would have to be cut out. A change that breaks verified functionality among such a large set of changes might not be noticed.
The original idea for approaching this issue was conditional compilation. Similarly to disabling `Display`/`Debug` for verification, Creusot `use` statements and annotations could be disabled when not verifying. The one major hurdle is the usage of Creusot-specific traits in `impl` blocks. For example, the start of the main `impl` block for `VecMap` when verified is in Listing 5.10. The Creusot-specific traits like `DeepModel` and `OrdLogic` cannot be disabled with conditional compilation. The idea for resolving this issue was supertraits. These are specific traits that require the implementation of other traits before the supertrait can be implemented. With this one could define a `Key` trait that is defined with conditional compilation to require Creusot-specific traits when verifying and to not have them regularly. To sketch the idea, it would look like in Listing 5.11. Unfortunately, this does not work, because associated type restrictions following the `where` clause on trait definitions in Rust's type system only restrict which types can implement `Key`, but when it is used for restricting generics, like in the `impl` block starting

```
pub enum Entry<'a, K, V>
where
    K: Ord + Eq,
{
    Vacant(VacantEntry<'a, K, V>),
    Occupied(OccupiedEntry<'a, K, V>),
}

pub struct VacantEntry<'a, K, V>
where
    K: Ord + Eq,
{
    map: &'a mut VecMap<K, V>,
    key: K,
    index: usize,
}

pub struct OccupiedEntry<'a, K, V>
where
    K: Ord + Eq,
{
    map: &'a mut VecMap<K, V>,
    key: K,
    index: usize,
}


```

Listing 5.12: The definition of the `Entry` type and members.

on line 10, it is ignored[5]. This means that once size comparisons are made with deep models for `Key`, the compiler complains that `OrdLogic` is not implemented, despite the definition saying otherwise. This means that the original idea of keeping one instance of the code for both verification and usage cannot be realised like this. Ultimately, the decision was made to have separate versions for verification and usage.

Verifying entry-getter functions

Insertion into `VecMap` is done through entry types. Such an entry can be obtained through `VecMap::entry`. An entry is either vacant (i.e. no value is associated with the key) or occupied. The entry type carries a mutable reference to the map during its lifetime, ensuring that it can only be mutated through the entry and nothing else. The `Entry` type is defined as in Listing 5.12. And the function for obtaining an `Entry` reads as in Listing 5.13. The issue here is that annotating this function with the post-condition `self.is_sorted()` will not verify in Why3, no matter what additional annotations are added. This is because `self` is passed into the `Entry` type and thus the function does

[5] `https://stackoverflow.com/a/47019398` is another example of this.

```rust
pub fn entry(&mut self, key: K) -> Entry<K, V> {
    match self.find_k(&key) {
        Ok(index) => Entry::Occupied(OccupiedEntry {
            map: self,
            index,
            key,
        }),
        Err(index) => Entry::Vacant(VacantEntry {
            map: self,
            index,
            key,
        }),
    }
}
```

Listing 5.13: The `VecMap::entry` function.

not own the reference to it when the function ends. The scope of `self` within Creusot's annotations does not exceed the function, which is why the previously presented post-condition does not verify. To do that, the map would have to be followed into the entry, and the annotation verified once `Entry` dies. This limitation means that the post-condition annotation is not possible for this function, contrary to what was required in §4.
But despite this limitation of Creusot, it is still possible to argue that the map remains sorted throughout the lifetime of `Entry`. The value of `map` is never passed on in `VacantEntry` or `OccupiedEntry`. This means that if it is possible to verify that the items in `map` stay sorted for every function call on the entries, it will also be sorted when those entries die. And this is in fact verifiable.
To demonstrate this with `OccupiedEntry`, first the `Invariant` trait is defined for it. It is defined in such a way that calling `Invariant::invariant` verifies that the map is sorted. This allows for checking that the invariant holds immediately after construction. This is done by adding the following post-condition to `VecMap::entry`:

```rust
#[ensures(forall<e: _> result == Entry::Occupied(e) ==> e.invariant())]
```

And finally, the invariant is also maintained for every function of `OccupiedEntry`, as seen in Listing 5.14. This means that `map` will be sorted when an instance of `OccupiedEntry` dies. The argument is analogous for `VacantEntry`. Thus it is proven that the entry-getter functions preserve the item order, despite Creusot's shortcomings.

Verifying `VecMap::entry_from_ref`

`VecMap::entry_from_ref` is in terms of functionality very similar to `VecMap::entry`.

```rust
impl<K, V> OccupiedEntry<'_, K, V>
where
    K: Ord + Eq + DeepModel,
    K::DeepModelTy: OrdLogic,
{
    #[maintains((mut self).invariant())]
    #[ensures((@(^self).map.v)[@self.index].1 == value)]
    pub fn replace(&mut self, value: V) {
        self.map.v[self.index].1 = value;
    }

    #[maintains((mut self).invariant())]
    pub fn get_mut(&mut self) -> &mut V {
        &mut self.map.v[self.index].1
    }
}
```

Listing 5.14: The annotated `OccupiedEntry`.

It returns the `Entry` matching to the supplied key. The difference is that a hint needs to be supplied as well, which indicates where an item for the key is in the underlying `Vec`. This function is defined in Listing 5.15. In essence, the map is searched linearly for `key` and `key_hint` skips items in the search, accelerating the process. This function was implemented for more performant getting of entries compared to `VecMap::entry` which does simply binary search. The function itself is not used in `vecmap` but is used once within the runtime. The problem with this function is that the sorting order of items may be invalidated if the correct input for `key_hint` is chosen. The type of the argument `KeyRef` is defined in Listing 5.16. It stores a key and the minimum index for where it is stored in the map. A `KeyRef` is valid if `key` is in the map and the location is stored in the map is greater or equal to `min_idx`.
There are two conditions that if met allow for compromising the sorting order of items with this function:

1. `key_hint` is invalid: while the validity of `key_hint` is checked at the start of the function, `debug_assert!` is removed from release builds. So if `key_hint.min_idx` refers to an entry with a key that is greater than `key`, then the function will return a `VacantEntry` allowing for inserting at that position with `key`. But this does not check for the possibility of keys greater than `key` before `key_hint.min_idx`.

2. `key_hint.key` > `key`: here the problem that is caused is the same. The difference is that `key_hint` may even be valid.

The easiest solution to this problem would simply be to remove this function and rely on `VecMap::entry` instead. But this would hurt the performance of the runtime which is not desired. So instead, these limitations have been properly documented and it was

```rust
pub fn entry_from_ref(&mut self, key_hint: KeyRef<K>, key: K) -> Entry<K, V> {
    debug_assert!(self.is_valid_keyref(&key_hint.as_ref()));
    let KeyRef { min_idx, .. } = key_hint;

    for i in min_idx..self.v.len() {
        match self.v[i].0.cmp(&key) {
            Ordering::Equal => {
                return Entry::Occupied(OccupiedEntry {
                    map: self,
                    index: i,
                    key,
                })
            }
            Ordering::Greater => {
                assert!(i >= 1); // otherwise min_idx
                return Entry::Vacant(VacantEntry {
                    map: self,
                    index: i,
                    key,
                });
            }
            _ => {}
        }
    }
    let i = self.v.len();
    Entry::Vacant(VacantEntry {
        map: self,
        index: i,
        key,
    })
}
```

Listing 5.15: The `entry_from_ref` function defined for `VecMap`.

```rust
pub struct KeyRef<K> {
    pub key: K,
    /// This is a lower bound on the actual index of key K,
    /// it doesn't need to be the index (though it usually will be).
    min_idx: usize,
}
```

Listing 5.16: The definition of the `KeyRef` type.

ensured that the one use of this function in the runtime, which is otherwise not covered by Creusot, is absent of the properties listed above.

5.2.4 Reproducing the verification

When `VecMap` was moved into its own crate, the file structure of the runtime was changed to this:

```
reactor-rs
├── runtime
│   └── …
└── vecmap
    ├── Cargo.toml
    ├── README.md
    ├── src
    │   └── lib.rs
    └── creusot
        ├── Cargo.toml
        └── src
            └── lib.rs
```

`vecmap` contains two crates, the one at the top-level for usage and the one in the `creusot` subdirectory for verification. As previously explained it was not possible to have one file for both purposes. During verification, the verifier is asked to use a diffing tool for `vecmap/src/lib.rs` and `vecmap/creusot/src/lib.rs`. The verifier should observe that between the two files, the bodies for Rust functions are unchanged, the only difference is that Creusot-specific annotations and traits were removed, as well as custom logic functions, and conditional compilation flags. If the function bodies are untouched, the verifier can be sure that the properties they are about to prove with Why3 do carry over to the version used in the runtime. The full verification process is described in `vecmap/README.md`.

5.2.5 Performance

Software verification often relies on constructs that compromise performance. Simple and naïve implementations are usually easier to reason about for verification than those optimised for performance. The verification attempt for the whole runtime in §5.1 is a good example of this. Creusot could not reason about unsafe code that was introduced to improve runtime performance, so, for verification, these optimisations were reverted. For verification work on `VecMap`, a more conservative approach was chosen. For example, instead of removing the problematic `VecMap::entry_from_ref` function in favour of the well-behaved but slower `VecMap::entry` function, it was kept to not hurt runtime

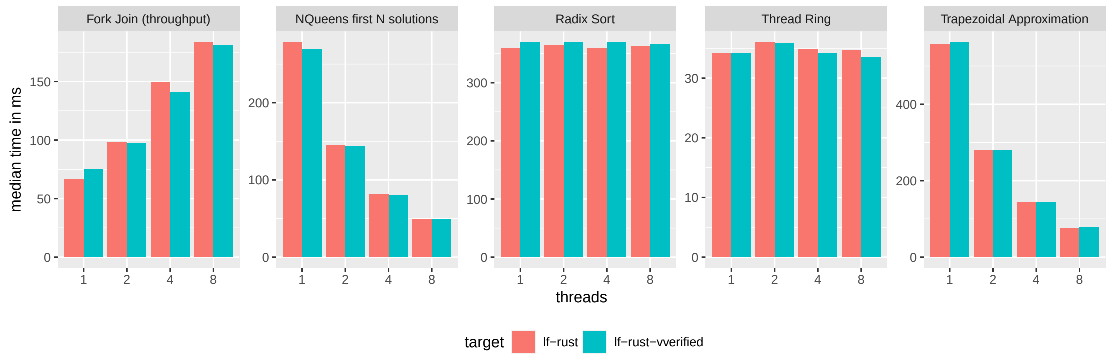


Figure 5.2: A comparison of the original Rust target performance from Figure 2.6 with the runtime version where `VecMap` is verified. Performance is effectively the same.

performance. But was this conservative approach successful? Figure 5.2[6] is affirming of this. Across multiple diverse benchmarks, there is no measurable drop in performance when the LF Rust runtime with the verified `vecmap` module is used.

5.2.6 Conclusion

Unlike the verification attempt for the whole runtime, this verification attempt for the `vecmap` module was a success. The smaller scope and lack of coupling within the code helped significantly. But the key difference is that `VecMap` does not use `impl`/`dyn` traits which do not work with Creusot. Other Rust features not supported by Creusot were a limiting factor for the extensiveness and productivity of this undertaking. But appropriate solutions were found.

[6]Reproducibility: Benchmarks performed on a machine with an Intel(R) Core(TM) i9-10900K CPU with 32 GB of RAM. The version of LF is the one identified by Git commit `7d52516` and for the benchmarking suite it is `dfba277`. For *lf-rust-vverified*, the runtime was substituted by the one identified by Git commit `5d61600`.

6 Conclusion

This work surveys the contemporary literature on Rust software verification and its tooling. Further, after detailing the Reactor Model and its implementation, Lingua Franca, correctness criteria for LF's runtime are given with a focus on the Rust target. The criteria were formulated to be easily translatable to what tools like Creusot require. An initial attempt at proving one of the correctness criteria, the absence of crashes, on the already existing codebase of LF's Rust runtime was made but unfortunately was not successful. The primary roadblock was the lack of support for some Rust language features by Creusot. The lack of support for concurrency and Unsafe Rust was anticipated and thus easy to work around. When standard library functionality used by the runtime was not covered by Creusot's contracts, those contracts were written and contributed back to Creusot. Among the other issues that were encountered, the lack of support for `impl` traits and working with external dependencies were the most severe, because no workaround is available. While the initial verification attempt was not successful, the verification of the important custom type `VecMap` within the runtime was. It shows that despite Creusot's limitations, already existing Rust software can be verified with relatively little modification if the to-be-verified code modules feature low coupling and avoid using language features not supported by Creusot. Still, formulating contracts that do verify, is a lot of work. Maintaining them within actively developed software modules would require arduous work, which means that ideal candidates for verification are those software components that are mostly or completely final.

6.1 Future work

There are many avenues for future work building on what was presented here. Most apparent is the further development of Creusot to lift the limitations enumerated in this work. Surely, this effort would benefit from continued research and development on other tools mentioned in this work or even the creation of wholly new ones. The landscape of Rust software verification is expanding quickly, and there are already efforts for adding official support for verification tools to Rust [Rus23]. During this work, the lack of support for `dyn` traits by Creusot was identified as the primary blocker for annotating the whole runtime for verification. If support for this language feature is added to Creusot in the future, another attempt at verifying the remaining correctness properties from §4 should be made. In the meantime, verification efforts for the runtime can focus on, for example, further analysing the composition of the runtime to see if some components can be decoupled from the rest of the runtime's code and individually verified, like `vecmap`.

Or, a similar verification effort could be made for external dependencies of the runtime. This work argued that usage of these libraries in other projects increases confidence that they are implemented correctly—but certainly would be good regardless. More broadly, regarding Lingua Franca, the language and its runtime should be stabilised, since there is little utility in verifying the functionality of a piece of software that might change soon in the future. Another avenue for future work would be to shift the focus from verifying the runtime to verifying the Lingua Franca compiler. By formally specifying the translation from LF source to the target language, it could be verified that a valid LF program always compiles to a valid and correspondingly specified program in the respective target language.

Bibliography

[Ast+22] Vytautas Astrauskas et al. ‘The Prusti Project: Formal Verification for Rust’. In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund and Ivan Perez. Vol. 13260. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 88–108. ISBN: 978-3-031-06772-3 978-3-031-06773-0. DOI: `10.1007/978-3-031-06773-0_5`. URL: `https://link.springer.com/10.1007/978-3-031-06773-0_5` (visited on 02/01/2023).

[Bar+11] Clark Barrett et al. ‘CVC4’. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: `10.1007/978-3-642-22110-1_14`.

[Bei16] Alexis Kenneth Beingessner. ‘You Can’t Spell Trust Without Rust’. Last Modified: 2016-04-05T08:54-04:00. Text. Carleton University, 2016. URL: `https://curve.carleton.ca/05076cd2-c1c2-4207-9667-a3ee1af58db4` (visited on 26/10/2022).

[BHR18] Marek Baranowski, Shaobo He and Zvonimir Rakamarić. ‘Verifying Rust Programs with SMACK’. In: *Automated Technology for Verification and Analysis*. Ed. by Shuvendu K. Lahiri and Chao Wang. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 528–535. ISBN: 978-3-030-01090-4. DOI: `10.1007/978-3-030-01090-4_32`.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. Cambridge, Mass. [u.a.]: MIT Press, 2008. ISBN: 978-0-262-02649-9.

[Bob+11] François Bobot et al. ‘Why3: Shepherd Your Herd of Provers’. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. 2011, p. 53. DOI: `10/document`. URL: `https://hal.inria.fr/hal-00790310` (visited on 11/01/2023).

[BOT21] Jim Blandy, Jason Orendorff and Leonora F. S. Tindall. *Programming Rust*. Second edition. OCLC: 1289839504. Cambridge: O’Reilly, 2021. ISBN: 978-1-4920-5254-8.

[CKL04] Edmund Clarke, Daniel Kroening and Flavio Lerda. ‘A Tool for Checking ANSI-C Programs’. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 168–176. ISBN: 978-3-540-24730-2. DOI: `10.1007/978-3-540-24730-2_15`.

[Con+18] Sylvain Conchon et al. 'Alt-Ergo 2.2'. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories. 12th July 2018. URL: `https://hal.inria.fr/hal-01960203` (visited on 11/01/2023).

[CPP18] CPP Reference authors. *std::vector<T,Allocator>::push_back*. cppreference. 9th Jan. 2018. URL: `https://en.cppreference.com/w/cpp/container/vector/push_back` (visited on 28/10/2022).

[DJ22] Xavier Denis and Jacques-Henri Jourdan. 'Specifying and Verifying Higher-order Rust Iterators'. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023. Preprint. 25th Oct. 2022. URL: `https://hal.archives-ouvertes.fr/hal-03827702/document`.

[DJM22] Xavier Denis, Jacques-Henri Jourdan and Claude Marché. 'Creusot: A Foundry for the Deductive Verification of Rust Programs'. In: *Formal Methods and Software Engineering*. Ed. by Adrian Riesco and Min Zhang. Vol. 13478. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 90–105. ISBN: 978-3-031-17243-4 978-3-031-17244-1. DOI: `10.1007/978-3-031-17244-1_6`. URL: `https://link.springer.com/10.1007/978-3-031-17244-1_6` (visited on 11/01/2023).

[Fou21] Clément Fournier. 'A Rust Backend for Lingua Franca'. Master's Thesis. TU Dresden, Dec. 2021.

[Jun+17] Ralf Jung et al. 'RustBelt: securing the foundations of the Rust programming language'. In: *Proceedings of the ACM on Programming Languages* 2 (POPL 27th Dec. 2017), 66:1–66:34. DOI: `10.1145/3158154`. URL: `https://doi.org/10.1145/3158154` (visited on 16/01/2023).

[KN19] Steve Klabnik and Carol Nichols. *The Rust programming language*. San Francisco: No Starch Press, 2019. 526 pp. ISBN: 978-1-71850-044-0.

[LAL18] Marcus Lindner, Jorge Aparicius and Per Lindgren. 'No Panic! Verification of Rust Programs by Symbolic Execution'. In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. 2018 IEEE 16th International Conference on Industrial Informatics (INDIN). ISSN: 2378-363X. July 2018, pp. 108–114. DOI: `10.1109/INDIN.2018.8471992`.

[Loh+20] Marten Lohstroh et al. 'Reactors: A Deterministic Model for Composable Reactive Systems'. In: *Cyber Physical Systems. Model-Based Design*. Ed. by Roger Chamberlain, Martin Edin Grimheden and Walid Taha. Vol. 11971. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 59–85. ISBN: 978-3-030-41130-5. DOI: `10.1007/978-3-030-41131-2_4`. URL: `http://link.springer.com/10.1007/978-3-030-41131-2_4` (visited on 18/10/2022).

[Loh20] Marten Lohstroh. 'Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems'. Issue: UCB/EECS-2020-235. PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html`.

[LR91] William Landi and Barbara G. Ryder. 'Pointer-induced aliasing: A problem classification'. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1991, pp. 93–103.

[Mat+22] Yusuke Matsushita et al. 'RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code'. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. San Diego CA USA: ACM, 9th June 2022, pp. 841–856. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523704. URL: https://dl.acm.org/doi/10.1145/3519939.3523704 (visited on 18/10/2022).

[MB08] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[MSS16] P. Müller, M. Schwerhoff and A. J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI).* Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2. URL: http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=MuellerSchwerhoffSummers16.pdf.

[MTK21] Yusuke Matsushita, Takeshi Tsukada and Naoki Kobayashi. 'RustHorn: CHC-based Verification for Rust Programs'. In: *ACM Transactions on Programming Languages and Systems* 43.4 (31st Dec. 2021), pp. 1–54. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3462205. URL: https://dl.acm.org/doi/10.1145/3462205 (visited on 26/10/2022).

[Nik17] Niko Matsakis. *Diving Into Rust For The First Time.* 10th Dec. 2017. URL: https://www.youtube.com/watch?v=_jMSrMex6R0 (visited on 01/11/2022).

[OGr22] Stephen O'Grady. *The RedMonk Programming Language Rankings: June 2022.* tecosystems. 20th Oct. 2022. URL: https://redmonk.com/sogrady/2022/10/20/language-rankings-6-22/ (visited on 27/10/2022).

[ORY01] Peter O'Hearn, John Reynolds and Hongseok Yang. 'Local Reasoning about Programs that Alter Data Structures'. In: *Computer Science Logic.* Ed. by Laurent Fribourg. Red. by Gerhard Goos, Juris Hartmanis and Jan van Leeuwen. Vol. 2142. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19. ISBN: 978-3-540-42554-0 978-3-540-44802-0. DOI: 10.1007/3-540-44802-0_1. URL: http://link.springer.com/10.1007/3-540-44802-0_1 (visited on 16/01/2023).

[Rusa] Rust community. *Implementations*. The Rust Reference. URL: `https://doc.rust-lang.org/reference/items/implementations.html` (visited on 07/02/2023).

[Rusb] Rust Lingua Franca runtime authors. *ReactionCtx*. reactor_rt. URL: `https://www.lf-lang.org/reactor-rs/reactor_rt/struct.ReactionCtx.html` (visited on 25/01/2023).

[Rus20] Rust community. *Appendix G - How Rust is Made and "Nightly Rust"*. The Rust Programming Language. 2020. URL: `https://doc.rust-lang.org/book/appendix-07-nightly-rust.html` (visited on 10/01/2023).

[Rus22] Rust language authors. *Overview of the Compiler*. Guide to Rustc Development. 5th Oct. 2022. URL: `https://rustc-dev-guide.rust-lang.org/overview.html` (visited on 01/11/2022).

[Rus23] Rust language team. *Rust Contracts RFC Draft*. 4th Jan. 2023. URL: `https://github.com/rust-lang/lang-team/blob/master/design-meeting-minutes/2022-11-25-contracts.md` (visited on 13/02/2023).

[Sko22] Sarek Høverstad Skotåm. 'CreuSAT - Using Rust and Creusot to create the world's fastest deductively verified SAT solver'. Accepted: 2022-09-20. Master thesis. 2022. URL: `https://www.duo.uio.no/handle/10852/96757` (visited on 14/11/2022).

[Van+22] Alexa VanHattum et al. 'Verifying dynamic trait objects in Rust'. In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE '22: 44th International Conference on Software Engineering. Pittsburgh Pennsylvania: ACM, 21st May 2022, pp. 321–330. ISBN: 978-1-4503-9226-6. DOI: `10.1145/3510457.3513031`. URL: `https://dl.acm.org/doi/10.1145/3510457.3513031` (visited on 01/01/2023).

[Why22] Why3 Development Team. *Why3 Documentation*. 12th Sept. 2022. URL: `https://why3.lri.fr/manual.pdf`.