Laboratory for Automated Reasoning and Analysis

Informal Systems

Master Thesis

Formal Verification of Rust with Stainless

Yann Bolliger

Company supervisor:	Romain Rüetschi, Informal Systems
Academic co-supervisor:	Georg Stefan Schmid, EPFL
Academic supervisor:	Prof. Viktor Kunčak, EPFL

July 30, 2021

Abstract

Writing correct software is hard, yet in systems that have a high failure cost or are not easily upgraded like blockchains, bugs and security problems cannot be tolerated. Therefore, these systems are perfect use cases for *formal verification*, the task of mathematically proving that a system conforms to its specification. Many recent blockchains are implemented in the Rust programming language because it guarantees memory safety at compile-time while providing full control over efficiency to the programmer.

To explore the powerful combination of the safety guarantees of Rust's compiler with the ability to formally verify high-level properties on programs, we present *Rust-Stainless*. Stainless is a formal verification tool for Scala backed by *Satisfiability modulo theories (SMT)* solvers. Rust-Stainless is a frontend for Stainless that extracts a subset of Rust, translates it to Scala and verifies it with Stainless. In this thesis project, I significantly advance the feature set of Rust-Stainless, increasing its expressiveness to programs that cannot be processed in Scala Stainless. In particular, this thesis adds support for mutability and in-place updates, by introducing a translation for mutability from Rust to Scala. I argue that the translation yields equivalent runtime semantics in both languages and refine it to work with Stainless. This allows to verify data structures implemented in Rust without sacrificing performance. Other new features are traits with verified contracts, references, and heap allocation.

The tool is evaluated on real-world Rust examples from a blockchain context. Remaining limitations concern missing support for some Rust language features, the limitation of only processing one crate at a time, and certain aliasing restrictions of the current Stainless backend for mutability. Finally, I explore possible promising ways for the future advancement of Rust-Stainless.

Contents

1 Introduction

Everyone who has ever written a computer program knows, it never runs correctly on the first try. Writing correct software is a hard task, yet the longer the more, software dominates our lives. While finding bugs in a student software project may be time-consuming but harmless, there are countless places where logical errors and worse, security problems cannot be tolerated. These are usually systems where failure has too high of a cost in terms of money or even human safety, imagine a railway control system. Software errors are also intolerable in systems that are costly or impossible to upgrade like embedded systems or satellites.

One type of systems that have both a high financial cost of failure and are difficult to upgrade are blockchains that power cryptocurrencies. Bitcoin [36] and Ethereum [12] have become very popular and are valued at amounts reaching into the hundreds of billions of dollars at the time of writing[1]. Upgrading such distributed systems is hard because a majority of the participating machines need to reach a consensus about the upgrade. This makes upgrading the two former blockchains nearly impossible and even systems designed with upgrades in mind like the Cosmos network [27] wish to minimise the number of upgrades required, hence the need for correct software.

A powerful and strict approach to writing correct software is to mathematically prove the correctness of a program, called *formal verification*. The Stainless verification framework [21] is a formal verification tool for the Scala programming language. It enables programmers to prove high-level properties about functions and data structures, for instance that they conform to their specification, in a semi-automated fashion. Additionally, Stainless establishes program termination and the absence of runtime crashes.

However, blockchains are usually not implemented in Scala, but rather in C++, Go or Rust for performance reasons. While Go achieves high performance despite its *garbage collector*, C++ and Rust leave memory management to the programmer. This is the primary source of security problems in C++. Rust, however, guarantees memory safety by introducing a new type checking phase in its compiler, the *borrow checker*. Rust is therefore well-suited for implementing highly performant, correct, and safe systems, for example the *Tendermint blockchain consensus* implementation[2] developed by Informal Systems.[3] But even with type safety and memory safety at compile-time, Rust cannot guarantee correctness on its own.

Therefore, and with the vision of combining the safety guarantees of Rust with the formally verified correctness guarantees of Stainless, we present *Rust-Stainless*, a verification tool created by Georg Schmid. Rust-Stainless[4] is a frontend to the Stainless verifier capable of extracting a subset of the Rust language, translating it to a subset of Scala and verifying it with Stainless. In this thesis project, I substantially extend the fragment of Rust that can be translated by the tool, adding features like mutability, references, and type classes.

[1]`https://coinmarketcap.com/historical/20210725/`
[2]`https://github.com/informalsystems/tendermint-rs`
[3]`https://informal.systems`
[4]Pun not intended, but – needless to say – welcome.

Before this thesis project, I mainly worked on the translation of Rust traits to Scala type classes, in the context of a semester project supervised by Georg Schmid. The thesis project was the best way to continue the ongoing work on Rust-Stainless. Therefore, this report also presents some of the results of the previous project. Thanks to Georg and Prof. Viktor Kunčak, my academic supervisor, I had the luck to conduct this thesis project as an intern at Informal Systems, where I was supervised by Romain Rüetschi. The blockchain- and Rust-focussed company provided the perfect motivation and context to push Rust-Stainless in the right direction.

Contributions

- **Theory**: In chapter 3, I develop a translation from Rust to Scala that produces equivalent runtime semantics in Scala for Rust features like mutability, mutable and immutable references, mutable tuples and data types, and move semantics. The translation is further adapted for use with the current state of Stainless's imperative phase.
- **Implementation**: The largest contribution of this project are the numerous features that I added to the implementation of Rust-Stainless. In over 70 pull requests, I fixed bugs, added extraction capabilities for new Rust language fragments, implemented the mutability translation, and improved the user-facing `stainless` library. Chapter 4 first describes the overall design and pipeline of the tool, then it outlines the state of the frontend before and after this project.
- **Bugfixes**: By using Stainless only as a backend and with unforeseen Scala-atypical inputs, we uncovered 14 issues in Stainless of which I solved or helped solve eight. Furthermore, I added the `freshCopy` primitive to the imperative phase of Stainless.
- **User Perspective**: The internship at Informal Systems allowed me to test our tool on real-world code, see subsection 5.2.2. This helped taking on a user perspective and had great influence on the choice and priorisation of new features.

2 Background

This chapter offers a short introduction to the Rust language as well as the Stainless verifier. Both parts will use a running example of a Rust program (Listing 2.25) to illustrate the involved concepts. It may seem odd to introduce Stainless with Rust as it is built and usually used with Scala. However, thanks to Rust-Stainless the usage of Stainless in Rust is so similar to the usage in Scala that it is suitable to reuse the running Rust example. For a more detailed introduction to Stainless in Scala please refer to its documentation [17].

2.1 Rust Language

Rust [25, 32] is a recent systems programming language, initially developed at Mozilla with the goal of replacing C++ as primary application language. Its most distinguishing feature is the new approach to memory management. Traditional languages tend to either manage memory with a *garbage collector* or require the programmer to manually allocate and, more importantly, deallocate memory. Rust deallocates memory itself without a garbage collector by using concepts of *ownership* and *reference lifetimes*. In other words, Rust provides memory management at compile-time which results in cost-free abstractions at runtime. Additionally, Rust guarantees memory safety properties like the absence of *dangling pointers*, i.e. it is impossible to dereference a freed part of memory.

To live up to its promises, Rust introduces a flow-sensitive type checker, the *borrow checker*, which is a concrete implementation of ideas that have been studied in research for years. The two main concepts are ownership [13] and *borrowing* that build on the research topics of *uniqueness* [11] and *linearity* [53, 49]. More details on the research background are in chapter 6.

2.1.1 Syntax Overview

Listing 2.1 shows a simple Rust function that returns the square of the integer `x`. Rust is statically typed. Function parameter types are annotated: `i32` in this case, a signed 32-bit integer. Other integer types can be unsigned (`u32`) or of other bit-lengths (`u128`). The return type of the function is indicated after the arrow, if omitted it defaults to `()`, the unit type. Note also, that the single (and last) expression in the function block is implicitly returned. Alternatively, one may employ the explicit return statement to return a value from any point in a function, like on line 3 of Listing 2.2. Statements are delimited with semi-colons and produce the unit type.

```rust
fn square(x: i32) -> i32 { x * x }
```

Listing 2.1: A simple Rust function.

Variables

The `let` statement is used to create a new variable and bind the assigned value to it. Function parameters are also variables in Rust [44]. By default, variables are immutable, that is, the compiler rejects any attempt to reassign to an initialised variable. On the other hand, Rust accepts to redeclare and thereby *shadow* a binding, which is what happens on line 5 of Listing 2.2.

The compiler infers the type of most let-bindings, such that the type does not have to be annotated like for function parameters. In Listing 2.2, all the types are inferred to be the standard `i32`. This even works for bindings that shadow an earlier declaration of a different type.

```rust
fn f(z: i32) -> i32 {
  let x = z * 10;
  if x > 100 { return z; }
  let y = z * 100;
  let z = 3;
  x + y + z
}
```

Listing 2.2: A Rust function doing some arithmetics.

Algebraic Data Types (ADTs)

Taking strong inspiration from C, user-defined data types in Rust are called `struct`s and `enum`s. Structs are product types and come in three forms: without fields, like `OnlyAMarker` in Listing 2.3, so called *unit structs*; the tuple form with numerically indexed fields, e.g. `a.0`; and the C-like structs with field names, `b.field`. Enumerations are sum types and have members that are again in either unit, tuple or C-like form. Listing 2.4 shows an enumeration type example from the standard library, the option type. It also shows how the type parameter `T` is used to create a generic data type.

```rust
struct OnlyAMarker;
struct A(i32, bool);
struct B {
  field: u8
}
```

Listing 2.3: All three forms of structs.

```rust
enum Option<T> {
    None,
    Some(T),
}
let opt: Option<i32> =
  Option::Some(123);
```

Listing 2.4: The standard option type as an example of a generic enumeration.

Rust also has anonymous tuples. These behave exactly like tuple structs but don't need to be declared beforehand. One can simply instantiate tuples of any non-negative number of types (cf. Listing 2.5). This is used in the first part of the running example (Listing 2.6) that is a generic container struct with an option of a tuple as field.

```rust
let t: (i32, bool) = (123, true);
```

Listing 2.5: A 2-tuple in Rust.

```rust
struct Container<K, V> {
  pair: Option<(K, V)>,
}
```

Listing 2.6: The struct for the running example.

Pattern Matching

With ADTs, especially enumerations, pattern matching comes naturally. Listing 2.7 shows how to match on the container struct from before. Like in Scala, the underscore is the wildcard pattern and match arms can be refined by `if` guards of any boolean expression. To merely check whether a value matches a certain pattern, one can use the `matches!(opt, Some(_))` macro that is expanded to a regular pattern match which returns true if the given pattern matches and false otherwise.

```rust
let c = Container { pair: None };
match c {
  Container { pair: Some((k, v)) } if k == 123 => Some(v),
  _ => None,
}
```

Listing 2.7: Pattern matching on a struct.

Mutability

As an imperative language, Rust also offers mutable bindings. Mutable variables have to be explicitly declared with the `mut` keyword. In general, Rust is very explicit about the distinction between mutable and immutable variables (and references). Listing 2.8 shows both a mutable function parameter and a mutable local variable.

```rust
fn g(mut a: i32) -> i32 {
  let mut b = 10;
  a += b;
  b = 100;
  a * b
}
```

Listing 2.8: Mutable variable bindings.

```rust
let mut b = B { field: 2 };
b.field = 123;
```

Listing 2.9: Mutable field of a mutable struct.

Fields are not variables in Rust but rather a part of their corresponding struct's variable [44]. Therefore, they inherit the mutability of their struct. For example, `b.field` in Listing 2.9 is mutable because `b` is mutable. Thus, mutability is not part of the field type but depends on the binding of the struct.

Implementations and Traits

Implementation blocks add methods to data types in Rust. There can be many such blocks for a given type. Listing 2.10 adds three methods to the container struct of the running example. The implementation block is generic in two type parameters `K, V`, as is the struct. Note that adding methods only for certain instantiations of the type parameters is also possible. For example, the first line of the block could be `impl Container<bool, i32>`, to only add methods to containers of boolean keys and integer values.

```rust
impl<K, V> Container<K, V> {
  pub fn new() -> Self { Container { pair: None } }
  pub fn is_empty(&self) -> bool { matches!(self.pair, None) }
  pub fn insert(&mut self, k: K, v: V) { self.pair = Some((k, v)) }
}
```

Listing 2.10: Methods for the running example.

The `Self` in the first return type stands for the type this block is adding methods to, in the example that is `Container<K, V>`. The `new` method is a static method, it is not called on instances of containers. In contrast, the two other methods specify a receiver type as first parameter. That makes the methods available on instances of the container, for example `c.is_empty()`. In methods with the `&self` receiver, the `self` keyword contains a reference to the instance on which the method was called, in the example of `c.is_empty()`, this is equivalent to `is_empty(&c)`. The same happens for `&mut self` but mutably.[1] Finally, it is also possible to consume the instance by specifying `self` or `mut self` as receiver.

Traits are used to specify interfaces in Rust. A trait `X` can be implemented for a data type `A` with an `impl X for A` block, providing implementations for the abstract methods of the trait, as in Listing 2.12. Traits can also be used to bound type parameters to types that implement a certain trait. For example, the trait bound `K: Id` in Listing 2.16 states that any `K` admissible to the function (or in other cases the block) needs to implement the `Id` trait, i.e. `impl Id for K` needs to be available.

```
trait Id {
  fn id(&self) -> isize;
}
```

Listing 2.11: Rust trait with one abstract method.

```
impl Id for String {
  fn id(&self) -> isize { 123456 }
}
```

Listing 2.12: Non-sensical implementation of the `Id` trait for strings.

Crates

Rust code is structured in *modules* that can be inlined in files, single files or entire folders. The unit of compilation in Rust is called a *crate* which is compiled to a binary or library. To import external code in a project, one has to import its crate, like `extern crate xx;`.

2.1.2 Owning and Referencing Data

Value Categories

Rust, like C but unlike Scala or any language on the *Java Virtual Machine (JVM)*, exposes memory management to the programmer and allows referencing stack-allocated data. Therefore, one needs to distinguish between the different *value categories* of an expression. The general definitions stemming from C are:

- *lvalues* are expressions that designate memory locations, for example variables, array elements etc. They are objects that have a storage address. The term originates from the fact that lvalues are on the left-hand side of an assignment.
- *rvalues* are expressions or temporary values that do not persist. They are on the right-hand side of an assignment [57].

These two terms were replaced in Rust by *place and value expressions.* Value expressions represent actual values and are defined by exclusion from place expressions. On the other hand, place expressions evaluate to a *place* which is essentially a memory location [44].

> [Place] expressions are paths which refer to local variables, [. . .] dereferences (`*expr`), array indexing expressions (`expr[expr]`), field references (`expr.f`) [. . .].
> [44, section "Expressions"]

Place expressions occur in *place expression contexts* like on the left-hand side of let-bindings, as borrow operands or as field expression operands. Otherwise, if a place expression is evaluated in a *value context*, e.g. on the right-hand side of a binding, it evaluates to what is stored at the place it designates. In other words, its data is *used.*

[1]More on references in subsubsection 2.1.2.

Ownership

As said previously, Rust's distinguishing feature is its ownership system which is responsible for:

> [enforcing] an ownership invariant where a variable is said to "own" the value it contains such that no two variables can own the same value [39, page 5].

In other words, data is owned by exactly one binding and if some data is used, e.g. in an assignment or passed to a function, it will be *moved out* of that binding, i.e. the ownership of the data is transferred to the new binding. The old binding is de-initialised and can never be reused. Moveable data behaves linearly [55], moving out can be seen as *destructive read* [22]. On line 4 of Listing 2.13, data is moved out of `a` and its ownership is transferred to `b`.

From the example it is clear, why Rust's borrow checker needs to be flow-sensitive. The move renders `a` dead before it formally goes out of scope, e.g. at the end of the block. The same applies to individual fields. A field can be moved out of its struct and become inaccessible while another field of the same struct is still available. This is called a *partial move*. The struct itself also becomes inaccessible as soon as one of its fields is moved.

```
1 let a = Container {
2   pair: Some(123, 456)
3 };
4 let b = a;
5 // `a` can never be used again
```

Listing 2.13: A struct type with move semantics.

```
let a = 123;
let mut b = a; // `a` is copied
b += 1;
// `a` can still be used
assert!(a == 123 && b == 124)
```

Listing 2.14: Copy semantics of the `i32` type.

Move vs Copy The semantics of moveable types are called *move semantics*. One departure from that are copyable types. Types that implement the `std::marker::Copy` trait get *copy semantics*. The compiler automatically implements that trait for primitive types[2] (cf. Listing 2.14) but also for shared references and tuples of copyable types [44, section "Special types and traits"]. When copyable data is used, it's implicitly copied bit-by-bit by the compiler and the original binding stays valid and independent. In contrast to shared references that are copyable, mutable references have move semantics because they need to stay unique, as will become clear shortly. Non-copyable types can still be explicitly copied by implementing the `std::clone::Clone` trait and calling the `.clone()` method.

Heap Allocation Up until now, all data was stack-allocated and hence, tied to the scope of the surrounding function. To outlive a function, data needs to be heap-allocated which is achieved in Rust with the `Box<T>` type. Everything that is put into a box is directly allocated on the heap (cf. Listing 2.15). The binding that "holds the box", `c_heap` in the example, is the unique owner of that data. Because boxes have move semantics, the ownership can be transferred to other bindings, e.g. across function scopes.

```
let c_heap = Box::new(Container { pair: None });
```

Listing 2.15: A heap-allocated container.

References

It is possible to read and even write data one does not own in Rust through references. In other words, references are the way to – temporarily – break away from the ownership invariant [39].

[2]Primitive types in Rust are booleans, characters, `str`, numeric types and the panic type `!` [44].

References are created by *borrowing* from a variable or field with the `&` or `&mut` operator, depending on the desired mutability of the created reference. One can either borrow from owned data directly or *reborrow* from an existing reference. Passing a reference to a function as an argument is called pass *by reference*, as opposed to passing the entire designated object *by value* and thereby moving or copying it. The method of Listing 2.16 receives a mutable reference to the container instance in `self` and returns a reborrowed mutable reference to the second part of its tuple.

```
impl<K: Id, V> Container<K, V> {
  pub fn get_mut_by_id(&mut self, id: usize) -> Option<&mut V> {
    match &mut self.pair {
      Some((k, v)) if k.id() == id => Some(v),
      _ => None,
    }
  }
}
```

Listing 2.16: An implementation block with a trait bound. The method returns a mutable reference to some interior part of the struct.

Rust's type and borrow checking system guarantees that all references are valid when they are used. That is, use after free errors are impossible and stack-allocated data cannot escape its allocation scope. To achieve this, Rust infers the *lifetime* of each reference, i.e. the part of the program in which the reference may still be used at a future point of any execution path [56]. A reference or variable that is not used anymore is said to be *dead*. The lifetime of a reference in Rust is constrained by the lifetime of the variable it borrows from. This is the main mechanism to prevent use after free problems. Furthermore, references need to adhere to a strict set of rules. At any point of a program and for any data there can either exist:

- multiple immutable, i.e. *shared references*, `&T`, that may read the referenced data,
- or a *unique mutable reference*, `&mut T`, that is allowed to read and write to the data it does not own.

During the lifetime of a shared reference, the borrowed data can be read but not changed by the owning binding. In contrast, the mutable reference allows to change data without owning it, under the condition that no other references to that data exist. While the mutable reference is live, even the owning binding is neither allowed to read nor to write to that place. In the words of Weiss et al. [55]:

> [...] we understand Rust's borrow checking system as ultimately being a system for statically building a proof that data in memory is either *uniquely owned* (and thus able to allow unguarded mutation) or *collectively shared*, but not both.

In-Place Updates It is common to implement mutable data structures in Rust with methods that alter the structure via a `&mut self` reference, for example `insert` in Listing 2.25. Such in-place updates are simple, if the method just needs to change some struct field, like in the example.

However, methods often need to temporarily consume some part of `self`, in order to change it. Imagine a method on `Option` that swaps a new value into a `Some` and returns the old option value. The naive approach to that method in Listing 2.17 does not compile because the ownership system does not allow moving out of a mutable reference, which happens on line 3. This makes sense because the mutable reference needs to stay valid, hence the data cannot disappear by moving out.

```rust
impl<T> Option<T> {
  fn swap(&mut self, t: T) -> Self {
    let tmp = *self;
    *self = Option::Some(t);
    tmp
  }
}
```

Listing 2.17: Naive swap implementation that does not compile.

```rust
impl<T> Option<T> {
  fn swap(&mut self, t: T) -> Self {
    std::mem::replace(
      self,
      Option::Some(t)
    )
  }
}
```

Listing 2.18: Swap using the `replace` method.

The solution is to use the special purpose function `std::mem::replace` that atomically replaces the value in a mutable reference with its second argument and returns the old value of the mutably borrowed place. For the `swap` in Listing 2.18, this is already the entire method.

2.1.3 Rust Compared to Scala

Mutability

Scala has local mutable variables (`var`) comparable to `let mut` in Rust. Class fields, however, are immutable by default. One can opt into mutability for a certain field by marking it as `var` at the class declaration. Such a field is then mutable on all instances of the class and irrespective of whether its corresponding instance is locally bound with the immutable `val` or the mutable `var`.

References

Places or lvalues are not programatically accessible in Scala, as on the JVM, there is no way to take a reference to a place explicitly. That means, it's impossible a priori to generalise over whether one wants to assign to a local variable or an object field by taking a reference to that lvalue/place.

As in all JVM languages, objects in Scala are heap-allocated and function parameters are always passed "by value". However, for objects this value is just a reference to the object in the heap. This is very efficient for sharing immutable objects while retaining by-value semantics. But as a consequence, the by-value passing of references closely resembles by-reference passing for objects, which is mainly important for mutable objects. The translation presented in chapter 3 will extensively exploit this fact.

2.2 Stainless Verifier

With a solid understanding of Rust, it is time to introduce verification. Stainless [21] is a formal verification tool for Scala. It lets programmers add contracts or specifications to functions and data structures. Stainless transforms the input program in multiple phases to a purely functional language understood by its backend *Inox* [52] which then tries to prove or disprove that the contracts hold by using Z3 [14] or a similar solver like CVC4 [6].

2.2.1 Specifications

The simplest example of a specification is shown in Listing 2.19. The expression in `require` is called a *precondition*, it has to hold when the function is called; the `ensuring` is a *postcondition* that must hold after the function if the precondition holds. Simply put, Stainless establishes just the latter: that the postconditions hold assuming the preconditions hold. Of course, Stainless is more sophisticated and additionally proves that the program finishes for all inputs (*termination*), that preconditions hold at static call sites of functions, that pattern matches are exhaustive, and that the program does not crash at runtime (except for out-of-memory errors) [17]. If these properties are invalid, then Stainless finds a counter-example input for the crashing or faulty function.

```scala
def fact(x: Int): Int = {
  require(x >= 0 && x < 10)
  if (x <= 0) 1
  else fact(x - 1) * x
} ensuring { r => r >= 0 }
```

Listing 2.19: The factorial with specifications in Scala.

```rust
#[pre(x >= 0 && x < 10)]
#[post(ret >= 0)]
fn fact(x: i32) -> i32 {
  if x <= 0 { 1 }
  else { fact(x - 1) * x }
}
```

Listing 2.20: The same factorial in Rust.

As Listing 2.20 illustrates, the same function and specification is also accepted by Rust-Stainless, which subsequently translates the program to Stainless and lets it prove its correctness. The only difference is that specifications are added as function attributes rather than inline statements and that postconditions in Rust automatically have a `ret` variable in scope that represents the return value. Otherwise, the two are very similar which, for simplicity, allows to introduce Stainless in the following sections only with Rust code examples. In both languages, one can put assertions into function bodies in addition to specifications, in Rust with the `assert!` macro. Stainless also proves that body assertions hold.

Up until now, this section has been imprecise about Stainless's input language. Stainless does not support the full Scala language but rather a functional subset called *PureScala*. The expressions in specifications need to be in PureScala. However, Stainless was extended to support some form of mutability [9] which this project relies on. Chapter 3 will go into details about mutability.

The methods of the container in the running example (fully displayed in Listing 2.25) show how the features from above are applied to add specifications. The `implies` function in line 26 is a library helper method provided by the `stainless` crate. Body assertions are used in the main function to ensure that the code behaves as expected.

Another library helper is the `old` function. This is very useful to refer to the value of a mutable reference before the function was executed in postconditions. For example in the postcondition of Listing 2.21, the `old(i)` will return the value of `i` before the function and `i` will have the value after the function. The helper can only be used in postconditions.

```rust
#[post(*i == *old(i) + 1)]
fn change_int(i: &mut i32) {
  *i = *i + 1;
}
```

Listing 2.21: The `old` helper for postconditions with mutable parameters.

2.2.2 Algebraic Properties

Traits are the way to specify mandatory interfaces in both Scala and Rust. Common properties like equality and ordering are often modelled with traits. While the languages can enforce implementors of traits to provide all abstract methods, they cannot enforce higher-level contracts. For example, for an equality trait with `eq` one would assume that all implementations are reflexive, i.e. $\forall x : x$ `eq` x. However, both compilers cannot guarantee that.

As a solution, one can specify contracts on traits in Stainless. First, trait methods can have specification attributes like regular functions. These will be proven to hold for all implementations. Furthermore, one can attach *laws* to traits, i.e. algebraic properties that implementors need to fulfil [17, section "Specifying Algebraic Properties"]. Stainless also proves the correctness of the laws for each implementation of the trait.

```rust
trait Rectangle {
  #[post(ret > 0)]
  fn width(&self) -> u32;
  #[post(ret > 0)]
  fn height(&self) -> u32;

  fn set_width(&self, width: u32) -> Self;
  fn set_height(&self, height: u32) -> Self;
  #[law]
  fn preserve_height(&self, any: u32) -> bool {
    self.set_width(any).height() == self.height()
  }
  #[law]
  fn preserve_width(&self, any: u32) -> bool {
    self.set_height(any).width() == self.width()
  }
}
```

Listing 2.22: Example trait with laws.

The last addition to the running example, is a law on the `Id` trait stating that the returned integer needs to be positive. Of course, the toy implementation that hard-codes an id for strings holds that contract (lines 7-8 in Listing 2.25). A more interesting example of laws in Listing 2.22 shows a violation of the Liskov Substitution principle [30]. The trait defines four methods. Two of them have regular postconditions and the laws state that a change to the width should not change the height and vice versa. Clearly, the implementation in Listing 2.23 conforms to these properties. Without laws, it would be tempting to add a square implementation, as in Listing 2.24. However, the square changes both dimensions at once – violating the laws. Stainless detects this, the square implementation fails verification, and thereby a possible bug is caught.

```rust
struct Rect {
  width: u32,
  height: u32
}
impl Rectangle for Rect {
  fn width(&self) -> u32 { self.width }
  fn height(&self) -> u32 { self.height }
  fn set_width(&self, width: u32) -> Self {
    Rect { width, height: self.height }
  }
  fn set_height(&self, height: u32) -> Self {
    Rect { width: self.width, height }
  }
}
```

Listing 2.23: Implementation of the rectangle trait.

```rust
struct Square {
  width: u32
}

impl Rectangle for Square {
  fn width(&self) -> u32 { self.width }
  fn height(&self) -> u32 { self.width }
  fn set_width(&self, width: u32) -> Self {
    Square { width }
  }
  fn set_height(&self, height: u32) -> Self {
    Square { width: height }
  }
}
```

Listing 2.24: Example implementation violating the laws.

This concludes the introduction to Rust and Stainless. All Rust features needed for the motivating example (Listing 2.25) were introduced, as were the verification features of Rust-Stainless. Most of these features were only added in the course of this project. The goal of the remaining chapters is now to extend Rust-Stainless with all these features. In particular, chapter 3 will explain how mutability and references are translated to Scala. Chapter 4 distinguishes between existing and added features, then gives some implementation details.

```rust
extern crate stainless;
use stainless::*;

trait Id {
  fn id(&self) -> isize;

  #[law]
  fn law_positive(&self) -> bool { self.id() > 0 }
}
impl Id for String {
  fn id(&self) -> isize { 123456 }
}

struct Container<K, V> { pair: Option<(K, V)> }

impl<K, V> Container<K, V> {
  #[post(ret.is_empty())]
  pub fn new() -> Self { Container { pair: None } }

  pub fn is_empty(&self) -> bool { matches!(self.pair, None) }

  #[post(!self.is_empty())]
  pub fn insert(&mut self, k: K, v: V) { self.pair = Some((k, v)) }
}
impl<K: Id, V> Container<K, V> {
  #[post((self.is_empty()).implies(matches!(ret, None)))]
  pub fn get_mut_by_id(&mut self, id: isize) -> Option<&mut V> {
    match &mut self.pair {
      Some((k, v)) if k.id() == id => Some(v),
      _ => None,
    }
  }
}

pub fn main() {
  let mut cont = Container::new();
  let id = 123456;
  let key = "foo".to_string();
  assert!(cont.is_empty());
  assert!(matches!(cont.get_mut_by_id(id), None));

  cont.insert(key.clone(), 0);
  match cont.get_mut_by_id(id) {
    Some(v) => *v = 1000,
    _ => panic!("no␣value"),
  };
  assert!(matches!(cont, Container { pair: Some((k, 1000))} if k == key))
}
```

Listing 2.25: The full running example for chapter 2 showcasing most of the newly introduced features of Rust-Stainless.

3 Mutability Translation

Although Rust introduces many ideas from functional programming into systems programming, it nevertheless is a language for high performance and low-level programming. Therefore, *idiomatic Rust* heavily uses mutability. To avoid having to rewrite Rust source code before being able to verify it with Rust-Stainless, the tool needs to support mutability. In this chapter, I introduce a translation of Rust's owned types and references to Scala that preserves runtime semantics. Further, I argue that the translation is correct and then present some changes and optimisations that make it possible to use the translation together with the current state of the Stainless verifier backend.

3.1 Translation for Runtime Equivalence

The translation works under the assumption that the input Rust code is from the *safe subset* [44, section "Unsafety"] of the 2018 edition of Rust and that execution is single-threaded. Further, the translation requires the absence of *interior mutability* [44, section "Interior Mutability"]. That means, all types are immutable through shared references. This can be enforced by prohibiting the use of *smart pointers* like the ones from `std::cell`. Moreover, the safe subset of Rust excludes features like raw pointers, `union`, unsafe functions and blocks, as well as external code, like linked C-code. In practice, all unsafe features are detected at extraction of the code and abort the translation.

3.1.1 Algorithm

The goal of the translation is to bridge the gaps between Rust's and Scala's memory model – making references accessible to the programmer and allowing mutability of all struct fields if needed. To achieve the first one and bring the ability of manipulating references to Scala, I introduce a special wrapper case class, the *mutable cell*:

```
case class MutCell[T](var value: t)
```

The mutable cell object is used as a layer of indirection on all Scala values and directly models the *place* of a value in Rust, where the value is stored in the field (`value`). Note that the field is mutable, which in principle allows changes to any place. After the translation, the mutable cell's field will even be the only mutable variable. All mutability in the original Rust program will be modelled with that field.

> Whenever something is (possibly) mutable, we wrap it into a mutable cell.

Local variables are wrapped into mutable cells, regardless of their mutability in Rust. This enables taking references to local variables which is needed for both mutable and immutable variables. The approach also allows to create references to data of primitive types like integers, something that is impossible to do in Scala directly.

Likewise, all `struct`, `enum`, and tuple fields are mutable cells with the type of the original field as the type argument of the cell. Thereby, the translation achieves its second goal from above – all fields are *possibly* mutable. In other words, every place in the Rust program is modelled by the introduction of a mutable cell object. Hence, all place expressions in the original program correspond to exactly one mutable cell instance in the translation.

```
struct A<T> {
  a: T, b: i32
}
let x = A {
  a: "foo", b: 123
}
let mut y = 123
assert!(y == x.b)
```

Listing 3.1: Example Rust struct.

```
case class A[T](
  a: MutCell[T], b: MutCell[Int]
)
val x = MutCell(
  A(MutCell("foo"), MutCell(123))
)
val y = MutCell(123)
assert(y.value == x.value.b.value)
```

Listing 3.2: The fields of the struct are modelled with mutable cells as are the two let-bindings.

Tuples

As for all structs, the mutability of tuples is decided at binding time of the instance in Rust. Contrary to Scala, where tuples are always immutable. To allow for tuple mutation in Rust, tuples of any positive arity are translated as case classes instead of Scala tuples. The following example shows such a class for a 2-tuple. The empty tuple corresponds to the unit type and is translated as such.

```
case class Tuple2[T0, T1](_0: MutCell[T0], _1: MutCell[T1])
```

Using Data

If a field or a variable is read, the translation introduces an access to the `value` field of the corresponding cell. The same happens for dereferencing. In other words, any place expression that is evaluated in a value context is translated as reading the `value` of the corresponding mutable cell object. This shows how the translation unifies the access of locally available bindings and references, that may come from elsewhere.

Copy vs Moving When place expressions are used in value contexts their data is moved or copied. For copyable types, the translation needs to ensure that the newly created copy is distinct from the original object. This holds trivially for primitive JVM types[1] as they are copied by value on the JVM as well. The same goes for shared Rust references, where the reference is copied but still points to the same object like on the JVM. However, tuples of copyable types, for example tuples of numbers, are copyable as well in Rust. This is a problem for the translation, because if it allows tuple objects to be shared by reference upon copy, the example in Listing 3.4 is incorrect in Scala.

To avoid such problems, the translation introduces two operators in Scala: `move[T](t: T): T` and `copy[T](t: T): T`. They can be thought of as top-level functions that perform a deep copy of their argument. An implementation for these operators could be based on type classes for example, but implementation details are left open here. Whenever data is used, the translation wraps the field access of the cell's `value` into one of the two operators, depending on the type of the data. For example on line 4 of Listing 3.4, the right-hand side of the assignment becomes `MutCell(copy(x.value))`. This is omitted for primitive JVM types as an optimisation.

[1]Numeric types, `chars` and booleans.

```rust
1 let x = (123, false);
2
3 // copies ‘x‘
4 let mut y = x;
5
6 y.0 = 456;
7 // holds:
8 assert!(x.0 == 123)
```

Listing 3.3: The tuple is copied on line 4.

```scala
1 val x =
2   MutCell(Tuple2(MutCell(123), MutCell(false)))
3 // shares tuple object ‘x.value‘
4 val y = MutCell(x.value)
5 // i.e. x.value == y.value
6 y.value._0.value = 456
7 // fails:
8 assert(x.value._0.value == 123)
```

Listing 3.4: In Scala, the tuple object is shared not copied on line 4. The solution to that is to add a `copy`. Hence, line 4 becomes `MutCell(copy(x.value))`.

User-defined copyable types, i.e. types for which the user derives a `Copy` implementation with the macro, run into the same issue as copyable tuples. The translation currently ignores these because the current state of our extraction implementation does not yet accept such types. Nonetheless, the solution would be to use the `copy` operator as well.

Referencing

A borrow in Rust creates a reference to a place and makes that reference available as a value in the program. After translation, the place expression in Rust is equivalent to the JVM cell object. Hence, passing around Rust references as values can be translated as passing around cell objects in Scala. (Passing around the cells conveniently happens by reference on the JVM.)

As an example, if a Rust reference is stored in a local variable, this creates two nested cells in the translation. The outer cell models the local variable and its field holds the other cell which models the Rust reference.

```rust
let x: T = ...;
let r = &mut x;
```

Listing 3.5: Taking a mutable reference in Rust.

```scala
val x = MutCell[T](...)
val r = MutCell(x)
```

Listing 3.6: Mentioning the cell `x` suffices in the translation.

Matching

The same principles apply for pattern matching as for evaluating place expressions. If a match happens on a value, the Scala match will be on the `value` field of that cell. If the scrutinee is a reference, then the Scala scrutinee is the cell object. The patterns are adapted to include the mutable cell wrappers of fields. Depending on whether a resulting binding of a match is a reference or not in Rust, the Scala pattern matches the cell or only its field, see Listing 3.8.

Assigning

Mutating values in both Rust and Scala is done by assigning new rvalues to lvalues. The translation for assignments is analogous to the one for accessing places and again unifies local variables and references. If a place expression is assigned an rvalue, the translation assigns to the `value` field of the cell, no matter whether the place expression is a local variable or a mutable reference, see Listing 3.10.

```
match x {
    A { a, .. } => ...
}
match &mut x {
  A { a, .. } => ...
}
match x {
  A { mut ref a, .. } => ...
}
```

Listing 3.7: In the first case, `a` binds the value of `x.a` whereas in the other cases it is `&mut x.a`.

```
x.value match {
    case A(MutCell(a), _) => ...
}
x match {
    case MutRef(A(a, _)) => ...
}
x.value match {
    case A(a, _) => ...
}
```

Listing 3.8: In the first case, the binding `a` matches `x.value.a.value` of the cell, whereas in the other cases it matches the cell object, `x.value.a`.

```
// `x` is of type `&mut i32`
*x = 123;
let mut y = 456;
y = 789;
```

Listing 3.9: Dereferencing and assigning in Rust.

```
// `x` is of type `MutCell[Int]`
x.value = 123
val y = MutCell(456)
y.value = 789
```

Listing 3.10: Assignments in Scala always go to the `value` field.

Function parameters

Rust functions can take parameters by reference or by value, see Listing 3.11. By-reference parameters are wrapped in a mutable cell in the translation. Additionally, function parameters are also variables and thereby places in Rust. To account for that and model the possible local mutability of the parameters (when marked with `mut`), the translation again wraps the parameter in a mutable cell, regardless of whether the parameter is `mut` or not. At function call sites, all arguments are wrapped into newly created cells.

```
fn f(
  mut i: i32,
  x: &i32
) { ... }
let r = 123;
f(456, &r);
```

Listing 3.11: `i` is only mutable inside the function.

```
def f(
  i: MutCell[Int],
  x: MutCell[MutCell[Int]]
) = ???
val r = MutCell(123)
f(MutCell(456), MutCell(r))
```

Listing 3.12: This results in nested cell in Scala.

Boxes

Heap allocation, i.e. boxes, are needed in Rust to create data that can outlive the function where it's created. Without boxes, data is always stack-allocated and thus dropped, at the end of the function. This distinction is not needed in Scala because, on the JVM, all objects live on the heap. Therefore, the translation can erase the fact that data is in a box, because it wraps the data in a mutable cell anyway to model the variable that "holds the box". Thus, the data is modelled on the heap which corresponds to what happens in Rust and thanks to the cell object all of this also works for primitive types.

```rust
fn f(a: &mut A) -> A {
  std::mem::replace(
    a,
    A { a: "bar", b: 0 }
  )
}
```

Listing 3.13: Replacing a mutable reference's value.

```scala
def f(a: MutCell[A]): A = {
  val res = move(a.value)
  a.value = A(MutCell("bar"), MutCell(0))
  res
}
```

Listing 3.14: The translation of the replace function.

Memory Replace

To allow for in-place updates of mutable references, the translation supports the Rust function `std::mem::replace`. With all the machinery introduced so far, the function can be translated as a simple feature on top of the `move` and `copy` operators, as shown in Listing 3.14. Interestingly, the translation corresponds closely to the implementation of the function – in unsafe Rust – from the standard library.[2]

3.1.2 Correctness

> Under the assumption that the original Rust program type and borrow checks, I argue that the translation of mutable Rust references and owned types to mutable cells in Scala is equivalent in runtime semantics, i.e. when run the Scala translation yields the same result as the Rust program.

The Rust borrow checker ensures that there is no aliased mutable data in the original program. This may seem like an unnecessarily strict assumption, but Rust's ownership system is so fundamental to the language that it is impossible to define Rust's semantics without borrow checking [54]. Consider Listing 3.15, because `y` moves out `x` on line 2, `x` can never be reused and its value is undefined, even if some code could illegally read it again. Therefore, the borrow check assumption is necessary to reason about the semantics of the original program.

```rust
1 let x = A { a: "foo", b: 123 };
2 let mut y = x;  // ‘x‘ is moved to ‘y‘
3 y.b = 456;
4 assert!(x.b == 123) // ERROR, can’t reuse ‘x‘
```

Listing 3.15: This code does not compile due to the use of `x` on line 4, after the move.

The translation introduces a bijection between Rust places and mutable cell objects. As seen before, places can be variables or fields. The translation wraps each local variable, each function parameter and each field into a mutable cell, hence, each of these places corresponds to exactly one cell. Arrays would differ from that but they are not yet supported.

As seen above, the translation unifies the access to values. Dereferencing and using local variables are both done through the cell object's field. One could say, the translation lifts everything into references; even local variables are accessed through the single gateway to the value that is the cell object. Furthermore, the cell's field is the only location where program data can reside. This corresponds to Rust's principle that data is always owned uniquely (by a cell) and otherwise referenced (through a cell).

[2]https://doc.rust-lang.org/src/core/mem/mod.rs.html#815

More formally, I examine in the following the two ways in which data can be accessed in Rust, by value or by reference, to argue that the translation results in correct Scala runtime semantics.

By value

If data is used by value, it grants ownership to the new binding. This holds for both moveable and copyable types. Only for the latter, the original binding and data stay intact, whereas for moveable types, the original owner is invalidated and the data is conceptually de-initialised (the compiler may invisibly optimise that away). In both cases, changes will not appear on the original binding if the used data is later mutated. Therefore, the translation is correct in inserting the `move` or `copy` operator – which for this argument can be taken as deep copies – around by-value reads of a cell's data.

Performing a deep copy for moveable types is correct because the original binding can never be reused. Thus, one can think of moving as performing a deep copy of the original object, assigning it to a new binding and then destroying the original object. Importantly, the changes made to the new, moved object never propagate back to the original.

For copyable types, the translation distinguishes between aggregate types and primitive types. Primitive JVM types are handled like in Rust and need no further examination. Aggregate types like tuples of numbers are translated to Scala objects with mutable cells as fields. Therefore, I need to show that the deep object copy the `copy` operator performs is equivalent to the implicit bitwise copy of the struct in Rust. Indeed, a bitwise copy for an aggregate type is trivially the same as a deep copy, as long as all contained types are present by value. That means, all contained data is embedded in the bits of the aggregate structure and is simply copied deeply (bitwise) as well. It gets more complicated for shared references (mutable references are not copyable as they need to be unique). Rust simply copies the shared reference, i.e. the pointer value, but not the referenced object. Still, that operation is equivalent to a deep copy for a shared reference because, by the borrow check assumption, as long as any shared reference to an object exists, that object cannot be mutated. In other words, a shared reference will never see its referenced value change under that assumption. Therefore, it is equivalent to copy that object deeply when needed because it cannot change. The sharing can be viewed simply as an optimisation in Rust.

The above holds for tuples, arrays and user-defined `Copy` types as long as the `copy` operator supports them in Scala. Other copyable types like function pointers are not supported by the translation nor the extraction.

By reference

The counterpart of using data and transferring its ownership is taking references to places. The translation handles that by using the place's cell object which is defined and unique, as shown earlier. Thanks to the cells, it becomes possible to reference places on the JVM; it suffices to mention, i.e. to use, the correct cell object whenever a reference is needed. Cells are JVM objects, hence, they can simply be used in a by-value manner and reference handling is done by the JVM. In particular, when a cell is used in multiple locations it will not be cloned but all the locations will point to the same cell object. This exactly corresponds to the semantics of shared Rust references that are model by the cells and is thus correct.

For mutable references, the mechanism is entirely the same. Any changes made on a cell's value will be visible to all readers, because they share the cell object. This sounds dangerous but the borrow check assumption guarantees that no illegal sharing or aliasing can occur. For example, it is guaranteed that all shared references to an object end their lifetime *before* the object is mutated. Thus, in the translation, a shared reference, in the form of a reference to a cell object, might still exist somewhere on the JVM when the cell is mutated and before the garbage collector passes, but it will never be read, thanks to the borrow checking. Therefore, the translation is correct for mutable references as well.

3.2 Translation for Stainless

The translation presented so far results in correct Scala runtime semantics, but the motivation was to ultimately use the translation to verify the Scala program with Stainless. Intuitively, the translated program can just be submitted to Stainless to get a verification result and deduce verification properties for the Rust program. That approach is correct because both programs exhibit the same runtime semantics. However, Stainless has some limitations concerning its support for mutability. Therefore, I adapt the general translation step-by-step to yield Scala programs that are accepted by Stainless.

This section presents the changes made to the translation to be compatible with Stainless at a theoretical level. Like before, I will argue that all the deviations from the general translation uphold the overall correctness. For more low-level implementation details and an overview of the entire system please refer to chapter 4.

3.2.1 Avoiding Aliasing

The main difference between full Scala and the imperative fragment supported by Stainless [17, section "Imperative"] is that Stainless imposes aliasing restrictions on the program. The restrictions permit a simple but strict aliasing invariant:

> [For] each object in the system, each path of pointers reaches a distinct area of the heap. [9, p. 59]

Concretely, the following restrictions apply to mutable types, i.e. types that may contain `var` fields or type parameters marked with `@mutable`:

- For a function call, no mutable arguments of the call can share any part of their memory. E.g. it is not allowed to give an object as one argument of a function call and a field of the same object as another argument.
- More generally, "it is forbidden to assign a mutable variable to a new identifier",
- and lastly, "a function cannot return an object that shares memory locations with one of its parameters" [9, p. 59].

As the translation uses mutable cells everywhere, virtually all resulting Scala types are mutable and fall under these restrictions. On the other hand, the motivation of Stainless's aliasing restrictions is to make its imperative code elimination phase work [9]. That transformation relies on the fact, that there is a single path to a mutable object. However, the restrictions are quite severe and there are plenty of cases where an alias is safe but is not allowed by these rules.[3]

In Rust, the borrow checker ensures a very similar property: there is a *unique* mutable reference to a place or otherwise it is safe to have multiple immutable references. The borrow check assumption guarantees that the program does not have illegal aliasing of mutable state, but the translation needs to convince Stainless of that fact. It does so with the `move` and `copy` operators. The translation already introduces `move` and `copy` when data is used by value, as can be seen in Listing 3.4 and Listing 3.14. To work around Stainless's aliasing restrictions, it additionally introduces `copy` for function arguments that are shared references and lastly, it adds `move` or `copy` to the return values of functions and inner blocks, unless the values contain mutable references. The correctness of that approach follows from the same argument as the correctness of using `move` and `copy` for using data by value (see subsubsection 3.1.2).

To summarise, all types are `move`'d or `copy`'d when they could create an alias except for types that are or contain mutable references. The borrow checker has made sure that the latter are

[3]For example, shared references, i.e. aliases, of a mutable object but the object is never changed.

not aliased. Hence, the resulting program contains no aliasing of mutable types but still preserves correct runtime semantics.

`move` is the Identity

This subsection shows that the `move` operator – though very useful to work with Stainless – is not necessary to the semantic correctness of the translation and may be omitted by implementations that do not need to comply with Stainless's aliasing restrictions.

> Under the assumption that the original Rust program type and borrow checks, the `move` operator is semantically equivalent to the identity and simply serves as a way to make verification work with a simple aliasing analysis.

As seen before, one can think of moving as performing a deep copy and invalidating the original. The copying during a move avoids creating an alias to the source cell. But, the borrow checker guarantees that a moved out value is never used again, hence the original will never be read again and its value does not matter for the correctness of the program, it might as well change.

It is easier to see the argument in the example of Listing 3.15. On line 2, the struct is moved out of `x` into `y`. That is, without the `move` operator, an alias of `x`'s cell would be created, but the binding of `x` will never be used again. Hence, `x` can be changed without problems and the inserted `move` only serves to communicate this fact to the simple aliasing analysis of Stainless. The same holds for function arguments and return values. Thus, in order to only achieve correct Scala runtime semantics, the `move` can be omitted because it is equivalent to the identity.

`copy` is the Functional Identity

The second claim only concerns completely functional programs, that is, programs that do not mutate any data.

> Under the assumption that the original Rust program type and borrow checks, and in the absence of *all mutation*, the `copy` operator is semantically equivalent to the identity and simply serves as a way to make verification work with a simple aliasing analysis.

Consider Listing 3.4, the introduced aliasing of the tuple object on line 4 is fixed by adding the `copy` operator, but the aliasing only poses a problem because the tuple is later modified. If the tuple wasn't modified, sharing would be safe and the program correct.

More generally, in the absence of mutation, there are only shared references and immutable by-value bindings. If data needs to be copied according to Rust semantics but is immutable, then it is safe to share the cell object in the translation because the data cannot be changed at any time. The same holds for shared references. Remember that a shared reference can never see its referenced data change, therefore it is safe to deeply copy the referenced object. But again, if that data cannot change at all, one may as well share the objects. This shows that the `copy` operator is equivalent to the identity in programs without mutation.

Cloning is Copying

Unlike the built-in implicit copy that occurs for copyable Rust types, a clone always happens explicitly by calling `clone()`. Programmers can define their own implementation of the `Clone` trait or they can let the `derive` macro generate one.

> Under the assumption that all `Clone` implementations in a program are derived by the macro, `clone` is equivalent to a deep copy and can be translated with the `copy` operator.

Derived `Clone` implementations for struct types recursively call `clone` on all the fields and return a new struct with the cloned fields. For copyable types, especially primitive types, the clone is trivially a bitwise copy like for `Copy`.[4] For shared references, subsubsection 3.1.2 shows that copying is equivalent to a deep copy of the referenced object.

By a simple inductive argument, if the `Clone` implementations of all the fields are equivalent to deep copies, then so is the struct's `Clone` implementation. Hence, as long as all component types recursively are copyable types or have derived `Clone` implementations, the top-level implementation is equivalent to a deep copy. The translation can therefore check that assumption and if it holds, translate calls to `clone` with the `copy` operator in Scala, which is equivalent to the identity in completely functional programs or performs a deep copy otherwise.

3.2.2 Optimisations

Even if the translation presented here serves to add mutability features to Rust-Stainless, it should preserve the well-functioning parts of the frontend that deal with immutable Rust. To that end, the translation that is implemented in software contains some optimisations that increase the usability of Rust-Stainless for functional Rust code. The following subsection presents these optimisations and argues that they uphold the overall correctness.

Immutable Bindings and References

The most important optimisation is that immutable bindings (local variables, function parameters) are not wrapped in mutable cells. Moreover, shared references are erased, like boxes, and their values are used directly. This optimisation is possible thanks to excellent support of the Rust compiler which provides type information for every expression. With that, it is possible to distinguish immutable values and shared references at all times from possibly mutable ones that are still wrapped in cells.

To convince the reader of the correctness of this optimisation, first note that only mutable data can be borrowed mutably. In other words, only for mutable data do we need the reference sharing in the translation, to propagate changes back to the original object. Data declared without `mut` is immutable and cannot change in the first place. Secondly, it is safe to directly use immutable values instead of references, even if the values get copied at some points, which again follows from the argument that one may deeply copy shared references (cf. subsubsection 3.1.2). Therefore, it is correct to erase shared references and use their referred values directly.

Preventing Aliasing

Previous sections showed how `move` and `copy` are used to emit code that complies with Stainless's aliasing restrictions. They also argued that `copy` *must* perform a semantic deep copy, while `move` *may* do a deep copy but might also be the identity. Our Stainless-backed implementation uses a newly added Stainless primitive to emulate both operators: `freshCopy[T](t: T): T` semantically performs a deep copy of its argument. It does not matter that many deep copies are introduced in the program in that way. The program is never run with these deep copies, they only serve for verification.

The implementation decides for each location where data is used by value whether to insert a `freshCopy` or not based on the type of the data. Types that are or contain mutable references,

[4] https://doc.rust-lang.org/std/clone/trait.Clone.html

```rust
1 let mut a = 123;
2 let mut b = 456;
3 let mut x = &mut a;
4
5 x = &mut b;
```

Listing 3.16: A mutable reference in a mutable variable.

```scala
1 val a: MutCell[Int] = MutCell[Int](123)
2 val b: MutCell[Int] = MutCell[Int](456)
3 val x: MutCell[MutCell[Int]] =
4   MutCell[MutCell[Int]](a)
5 x.value = MutCell[Int](b)
```

Listing 3.17: Translation with nested mutable cells.

like `Option<&mut T>`, are not `freshCopy`'d and the copy is also omitted for primitive JVM types. Additionally, the translation avoids `freshCopy` around some control structures for which it is clear by design that a `freshCopy` will be inserted inside the structure. For example in a `match` expression, the resulting value of each arm will be copied and therefore the entire match expression doesn't need to be wrapped in an additional `freshCopy`.

The same procedure is applied for cloning. Conveniently, the Rust compiler does not derive `Clone` implementations for types that are or contain mutable references.[5] Therefore, it is sufficient to check that all `Clone` implementations are derived (cf. subsubsection 3.2.1) to ensure that all types for which `clone` is translated will be `freshCopy`'d.

To summarise, the implementation deeply copies some of the moves and all the copies that happen in Rust to prevent aliasing in the resulting code. But, it never copies mutable references because for those, the Rust compiler has made sure that there are no aliases. Lastly, the same happens for cloning because, by the check, only clone operations are extracted for which these semantics are correct.

3.2.3 Limitations

So far, mutable cells together with the `freshCopy` primitive solved all cases of aliasing restriction problems. However, there are patterns where that solution is not applicable and these constitute the theoretical limits of this translation with the current aliasing restrictions of Stainless – the runtime equivalence in full Scala is still valid.

The unsupported patterns occur when mutable references are used in locations where they cannot be `freshCopy`'d to keep the semantics correct. For example, a mutable binding of a mutable reference: `mut x: &mut T`. The variable `x` can be dereferenced and assigned `*x = ...`, which changes the referred object, but it can also be reassigned with another mutable reference `x = &mut ...`. To correctly model both of these cases, the translation gives the type `MutCell[MutCell[T]]` to `x`, like in Listing 3.17. Note that the example runs correctly in Scala. Unfortunately, the same cannot be said about Stainless. The assignment on line 5 creates an alias of `b` in `x.value`, which Stainless does not permit. This can occur for local variables as in the example but also for `mut` function parameters of mutable reference type.

The other case of mutable references, i.e. mutable cells, that cannot be `freshCopy`'d occurs in function return values. In the running example of Listing 2.25, `get_mut_by_id` returns a mutable reference. This is correctly translated and also verified. However, Stainless does not allow the same for recursive functions, because of the third aliasing rule in subsection 3.2.1. Thus, the same `get_mut` method for a list which is recursive, shown in Listing 3.18, is rejected by Stainless.

[5] https://doc.rust-lang.org/std/clone/trait.Clone.html#impl-Clone-121

```rust
impl<V> List<(u128, V)> {
  pub fn get_mut(&mut self, key: &u128) -> Option<&mut V> {
    match self {
      List::Nil => None,
      List::Cons(head, _) if head.0 == *key => Some(&mut head.1),
      List::Cons(_, tail) => tail.get_mut(key),
    }
  }
}
```

Listing 3.18: A recursive method returning a mutable reference.

This chapter presented a translation for mutability from Rust to Scala and refined it in several steps to make it useable with the current imperative phase of Stainless. The general translation is quite powerful and results in equivalent runtime semantics in Scala for mutability features of Rust. For each version of the translation, I argued that the runtime equivalence is upheld. The last section discussed the theoretical limitation of the approach. Further limitations more associated with the implementation of this translation are discussed in section 4.4. More generally, the next chapter sheds light on the engineering of Rust-Stainless and other new features than mutability.

4 Implementation

Rust-Stainless is developed as an open source software project.[1] It was created by Georg Schmid to explore a new frontend to Stainless for the Rust language. Upon completion of this thesis project, the tool is implemented in approximately 12K lines of Rust code. Most features described here are available on the `master` branch, except for the mutability translation that lives on the `mutable-cells` branch.[2]

Before presenting the tool's design and implementation details, the next section provides some background on the Rust compiler needed to understand the inner workings of the tool. The subsequent sections distinguish between language features the tool was already capable of extracting before this project and new features that have been introduced. The final section discusses limitations of the current implementation.

4.1 Background

4.1.1 Rust Compiler

To translate Rust without too much busy-work, Rust-Stainless leverages the heavy lifting done by the Rust compiler [43], `rustc`. The compiler takes Rust source code as input and produces an executable binary. This process involves multiple phases that each transform or *lower* a higher-level input to a lower-level representation. The following paragraphs give a brief overview of the compilation process and the leftmost column of Figure 4.1 illustrates the different phases.

The Rust compiler always operates on a single crate. That is, each crate is compiled separately and only afterwards linked to potential other crates. This means that the *intermediate representations (IRs)* described hereafter are only available for items of the currently compiled crate with some exceptions for metadata of public items, like function signatures that need to be accessible across crates.

The first two compilation phases transform the input text into a token stream – lexing – and then into an *abstract syntax tree (AST)* – parsing. First syntactical validations, name resolutions and the macro expansion are performed on the AST. While the AST is already a representation of the program, it still maps directly to the source code and can be seen as a data structure representation of the source text.

The first intermediate representation constructed from the AST is the *high-level IR (HIR)*. It contains less high-level features than surface Rust, because many have been desugared. The HIR is used to type check the program. In that process another IR is created, the *typed HIR (THIR)*. Still in the form of a syntax tree, the THIR contains fully explicit type information.

The THIR is used to construct the *mid-level IR (MIR)* that has the form of a *control-flow graph (CFG)*. Such a graph is a diagram that consists of *basic blocks* of code and arrows between them

[1] `https://github.com/epfl-lara/rust-stainless/`

[2] The commits on the two branches corresponding to the state at the end of this project are:
master: `https://github.com/epfl-lara/rust-stainless/commit/1e16201c0b63fcc7f8871f0f9e9974b663e0e3eb`,
mutable-cells: `https://github.com/epfl-lara/rust-stainless/commit/1df1d28d6100a2306a8cd5ed1b4445c60512d9b2`,
PR of the mutability translation: `https://github.com/epfl-lara/rust-stainless/pull/164`.

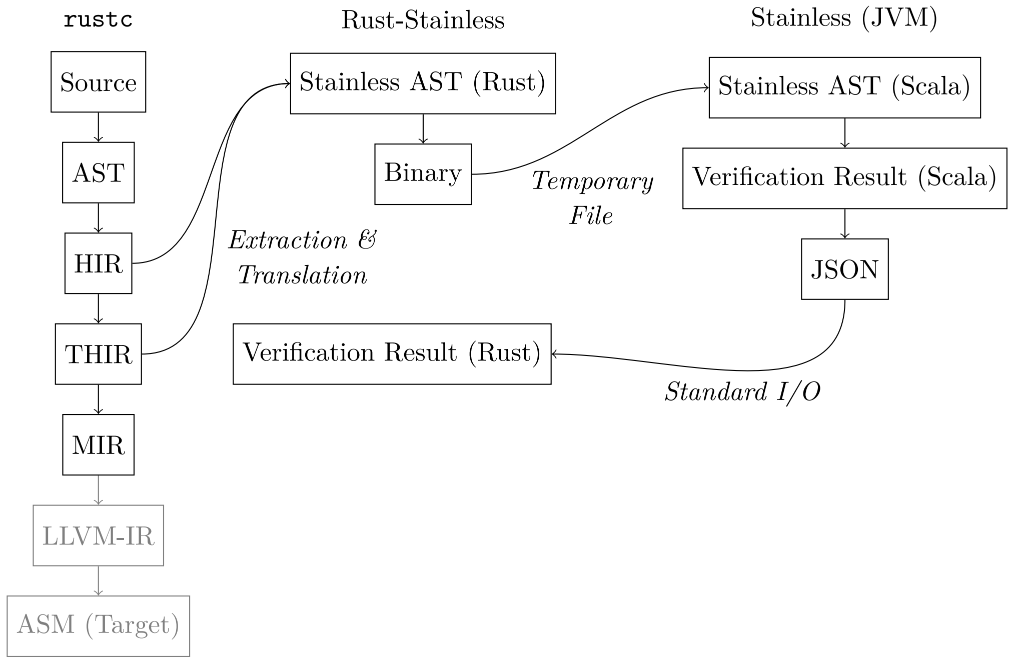

Figure 4.1: The Rust-Stainless pipeline represented by all intermediate representations of the program. The left column on its own shows the normal compilation pipeline of Rust. The grey parts are not executed in Rust-Stainless.

that represent all the possible paths the control flow can take. The MIR still has generic types but because it is a CFG, it is much more convenient for processes like liveness analysis, optimisations, and most importantly, the borrow check of the program.

The last compilation phase is *code generation*. Generic code is *monomorphised*, i.e. copied and specialised for each type it is instantiated with. Then, a special IR for *LLVM* [28] is generated. LLVM is a general purpose compiler backend, also used by C++. It will perform many more performance optimisations and finally generate the assembly code for the target architecture.

4.2 System Overview

4.2.1 Design

Our tool consists of multiple components, most of which are implemented as their own crate. Rust-Stainless has itself a frontend and a backend. `stainless_frontend` contains two executables that start the tool and deal with command line arguments. The subcommand for the Cargo build tool, `cargo stainless`, is the most common way of running Rust-Stainless. Internally, it calls the second stand-alone binary, `rustc_to_stainless`, which runs the actual frontend.

The programmer also sees another interface of Rust-Stainless, its library `libstainless`. The library needs to be imported as `extern crate stainless;` in all code to be verified. It provides the user-facing parts of Rust-Stainless like the specification macros and some built-in Stainless types useful for specification.

The principal translation is performed by the `stainless_extraction` crate. The frontend calls the method `extract_crate` which retrieves the HIR from the compiler and translates it to Stainless AST with the help of another crate, `stainless_data`. The latter contains auto-generated Rust definitions of the Stainless AST types as well as code to serialise them.

Finally, `stainless_backend` is responsible for spawning and interacting with a JVM subprocess of the Stainless executable. The executable consist of the normal Stainless verification pipeline but with a custom entry-point called `Noxt-Frontend`. “Noxt” stands for *no extraction*; it takes serialised trees as input instead of extracting trees from the Scala compiler.

4.2.2 Pipeline

Having introduced the system’s components, the next section traces the path of an exemplary program through the pipeline in more detail. The path can also be followed in Figure 4.1. Gaining insight into the pipeline will increase the understanding of the design choices made.

In Listing 4.1, the Stainless library is imported such that the postcondition specification attribute `#[post(...)]` is available. The attribute is implemented as a procedural macro[3] and will be expanded to a closure inside the function (Listing 4.2). More details on the specification encoding follow in subsubsection 4.3.2.

```rust
extern crate stainless;
use stainless::*;

struct A(i32);

#[post(ret.0 >= 0)]
fn f(a: A) -> A {
  A(a.0 * a.0)
}
```

Listing 4.1: Rust program with a postcondition.

```rust
fn f(a: A) -> A {
  #[clippy::stainless::post]
  |a: A, ret: A| -> bool { ret.0 >= 0 };
  A(a.0 * a.0)
}
```

Listing 4.2: Desugared postcondition.

The `stainless_frontend` invokes the compiler via the `rustc_driver` library and lets it run until all analyses are completed and have passed, otherwise the tool fails with the error returned by `rustc`. In particular, the compiler does the lexing, parsing, macro expansion, and IR construction. Secondly, it type and borrow checks the program. This is how the strong assumption from subsection 3.1.2 is achieved. Once analysis is complete, the extraction module is invoked on the HIR. As the HIR is only ever constructed for the current crate, Rust-Stainless also only translates the current crate. This limitation of the tool is discussed in section 4.4.

The extraction traverses all the *items* of the crate, that is, top-level functions, structs, enums, `impl` blocks with methods, as well as traits. For ADTs, the extraction directly constructs Stainless definitions. For function bodies, it uses the THIR, which is guaranteed to exist because the type check passed. For illustration, Listing A.1 shows the THIR for the body of the function in Listing 4.1.

```scala
sealed case class A(_0: MutCell[Int])
@synthetic sealed case class MutCell[T @mutable]((value: T @mutable) @var)

@pure
def f(a: A): A = {
  freshCopy(A(MutCell[Int](a._0.value * a._0.value)))
} ensuring {
  (ret: A) => ret._0.value >= 0
}
```

Listing 4.3: Extracted Stainless ADTs and functions, printed as code.

[3]Procedural macros invoke user-provided code at compilation-time and thus allow for more complex transformations of the AST inside a macro [44, section “Procedural Macros”].

If all the THIR bodies are translated to Stainless AST without errors, i.e. there are no unsupported features in the program, the Stainless AST is serialised to a custom binary format understood by Stainless. At this point, the program is represented by a list of functions, a list of ADTs and a list of classes (Listing 4.3).

If the user specified to export the AST with `--export`, the binary format is simply written to a file. Otherwise, Rust-Stainless spawns a subprocess with the `NoxtFrontend` of Stainless. The subprocess reads the binary format from a temporary file, performs minor transformations and passes it to the verification pipeline. In the end, it reports back the results in JSON format via standard output such that Rust-Stainless can print them. In this example, Stainless returns a problem, see Listing 4.4: the integer multiplication on line 6 of Listing 4.3 could overflow and therefore, the postcondition does not hold.

```
$ cargo stainless --example ex
cargo-stainless: Found example target 'ex'.

=== Analysing crate 'ex' ===

[ Discovering local definitions ]
  - ADT A
  - Fun main
  - Fun f
[ Extracted items ]
 - ADT        A$0
 - ADT        MutCell$2
 - Function   main$5
 - Function   f$6

note: Verified 2 items.
warning: Failed to prove 1 VCs:
- f              postcondition                 0.3              Invalid

warning: 1 warning emitted
```

Listing 4.4: Console output of `cargo stainless` for the code of Listing 4.1.

4.3 Extraction

4.3.1 Supported Rust Features

Existing Features

The fragment of Rust that the tool could extract and translate before this project underlay strict restrictions: all code needed to be functional, immutable, and the only allowed side-effect was `panic!`. References, heap allocation and with it recursive data types were unsupported. Apart from that, a large part of the language was already supported, i.e. most of the control-flow syntax, top-level functions with bodies, integer and boolean expressions, string literals, pattern matching, ADTs, including tuples (without their pattern matching), and generics. Function specifications (specs), that is pre- and postconditions, could be stated with the `pre` and `post` attributes from the `stainless` crate.

The expression in a spec is a regular Rust expression that must have no effect on any variables of the function body. This posed a problem in the absence of references because oftentimes, the

```rust
pub fn i32_ops(x: i32, y: i32) {
  assert!(x + y == 2 * x);
  if x >= 0 && x < 1<<30 {
    assert!(x == (x + x) / 2);
  }
}
enum Maybe<T> {
  Nothing,
  Just { value: T }
}
fn get_or<T>(maybe: Maybe<T>, default: T) -> T {
  match maybe {
    Maybe::Nothing => default,
    Maybe::Just { value } => value,
  }
}
#[pre(x >= 0)]
#[pre(x < 10)]
#[post(ret >= 0)]
pub fn fact(x: i32) -> i32 {
  if x <= 0 { 1 }
  else { fact(x - 1) * x }
}
```

Listing 4.5: Example of existing features in Rust-Stainless.

expression would consume a function parameter of moveable type multiple times, which does not borrow check. As a work-around, one could add multiple specs of the same kind to a function, which is equivalent to multiple `&&`-concatenated expressions. With all of the above, an example of supported and verified Rust code before this project is Listing 4.5.

New Features

This section gives an overview of all the Rust language features that were added to the extraction of Rust-Stainless in the course of this project except for the mutability translation which has already been introduced in chapter 3. The features described here are available on the `master` branch of the project.

Syntactical and Notational Improvements Support was added for:

- `else if` expressions,
- `let` bindings with user-specified type annotation,

  ```rust
  let t: u16 = 1;
  ```

- pattern matching on tuples,
- accessing tuple struct fields by their numerical identifier, `A(2, 3).0`,
- the `return` keyword at most points of a function (but not in `if` conditions and guards),
- `usize` and `isize` integer types that have the bit-length of a pointer on the target platform,

- *struct update syntax*, a short-hand notation for creating a struct from an existing one,

  ```
  struct A { a: i32, b: bool, c: char }
  let x: A = A { a: 123, b: true, c: 'c' };
  let y: A = A { b: false, ..x }; // copies `x.b`, `x.c`
  ```

- crate-local modules and imports, which enables splitting up a crate into multiple files, and finally,

- panics in expression locations like in a pattern match arm.

  ```
  match Option::Some(123) {
    Option::Some(x) => x,
    Option::None => panic!("no␣value"),
  }
  ```

Immutable References and Heap Allocation It is now possible to immutably borrow places, pass immutable references as values, and allocate data on the heap with boxes. Although already presented in chapter 3, the feature is mentioned here because it exists independently of the mutability translation.

The above enables recursive data types like the typical, functional linked-list (see below). Additionally, it eases the problem of borrow checking spec expressions because expressions that only read data, like most specifications do, can now take a reference instead of consuming the data.

```
pub enum List<T> {
  Nil,
  Cons(T, Box<List<T>>),
}
```

Measure Attribute Inductive proofs in Stainless often require the programmer to state the induction variable with a `decreases` call in Scala. This helps Stainless infer the so called *measure* of the proof, with which it checks termination. The same feature was introduced in Rust as a new spec attribute that enables verification of recursive functions, like Listing 4.6.

```
#[measure(l)]
fn size<T>(l: &List<T>) -> u32 {
  match l {
    List::Nil => 0,
    List::Cons(_, tail) => 1 + size(tail),
  }
}
```

Listing 4.6: Measure attribute.

Stainless Library The `stainless` crate is exposed to the programmer and contains helpers for specifying proofs and conditions. This is the equivalent of the `stainless` package in Scala. In addition to the spec attributes, `libstainless` now offers an immutable, infinite set `stainless::Set<T>` and map `stainless::Map<K, V>`, see Listing 4.7.

In extraction, both types are translated to Stainless's built-in infinite set and map type. Hence, they are backed and well-understood by Stainless which enables their proof utility. At runtime, the collections are backed by a runnable implementation that relies on the `im`[4] crate. The Rust

```rust
Set<T> {
  fn new() -> Self;
  fn singleton(t: T) -> Self;
  fn insert(&self, t: T) -> Self;
  fn contains(&self, t: &T) -> bool;
  fn union(self, other: Set<T>) -> Self;
  fn intersection(self, other: Set<T>) -> Self;
  fn difference(self, other: Set<T>) -> Self;
  fn is_subset(&self, other: &Set<T>) -> bool;
}
Map<K, V> {
  fn new() -> Self;
  fn get(&self, key: &K) -> Option<&V>;
  fn get_or<'a>(&'a self, key: &K, elze: &'a V);
  /// Panics if the key is not in the map.
  fn index(&self, key: &K) -> &V;
  fn contains_key(&self, key: &K) -> bool;
  fn insert(&self, key: K, val: V) -> Self;
  fn remove(&self, key: &K) -> Self;
}
```

Listing 4.7: The interface of the Stainless collections in Rust.

interface of the collections was designed to resemble the `std::collections::HashSet` and `HashMap` as closely as possible.

Furthermore, the library provides a helper function `implies` that lets one write the logical implication $p \implies q \equiv \neg p \wedge q$ over boolean expressions. The library also offers the `old` helper for postconditions that returns the value a mutable variable had before the function.

External and Synthesised ADTs Internally, the `Map<K, V>` of the Stainless crate relies on the `MutableMap` of Scala Stainless with values of type `Option[V]`. To perform that translation, the extraction needs to create trees of the `Option` type, even if the `std::option::Option` doesn't occur in the program. Therefore, Rust-Stainless can now synthesise values and types for certain specific ADTs: `Option`, tuples and mutable cells as described in section 3.1.

The frontend also supports extraction of crate-external ADTs like `std::result::Result`. That means, the programmer can use the standard structures and they are correctly translated, provided that no methods of these types are used. This is the one-crate-limitation, discussed in section 4.4.

Implementation Blocks, Traits and Laws The largest addition, other than mutability, of this project is support for implementation blocks, methods, and traits with contracts, as introduced in subsection 2.2.2. This includes solving some intricate problems. For example, the Rust compiler automatically and implicitly resolves which trait implementation to use for any given trait method call. If no suitable implementation is in scope, it signals an error. A similar mechanism can be achieved in Scala by means of *type classes*. The resolution of type class methods and instances is done through the *implicits mechanism* in regular Scala. In Stainless however, this resolution has to be done manually. Therefore, Rust-Stainless not only extracts type classes from Rust traits but also infers which type class instance to call at each method call site.

Furthermore, our tool is capable of extracting laws and specifications of traits and their implementations, including specification attributes on abstract methods. Lastly, it can extract trait

[4] https://docs.rs/im/15.0.0/im/

bounds like `T: Equals` from top-level functions and implementation blocks, and transform them to the equivalent Scala pattern of *evidence parameters*. More details on that translation follow in the next section.

4.3.2 Implementation Details

Extraction Overview

As seen in subsection 4.2.2, the majority of the work of Rust-Stainless happens in the extraction of the HIR to Stainless AST. This subsection takes a more detailed look at the most interesting phase of our tool, implemented in the `stainless_extraction` crate.

Before dealing with user code, the extraction needs to register some specially treated items, called *standard items*. These are Rust language features like panic, standard library items like `Box<T>` and `Option<T>`, but also all items of `libstainless` like `Set<T>` and `implies`. Extraction needs to know the identifiers (the `DefIds`) of those items in order to recognise them in user code and trigger their specialised treatment. Unfortunately, these `DefIds` are not stable across compilation runs and therefore, the detection needs to happen at each run of the tool.

Initially, only the *definition paths* of the items to detect are known,[5] but that does not suffice because `rustc` does not have an API to query items from other crates *by path* at the time of writing. All the lookups are done by `DefId`. Therefore, the implementation has to enumerate all `DefIds` from the `std` and `stainless` crate, compare them by name to the desired items, and register the required ones. This approach is clearly a brute-force work-around to the lack of by-path lookups in `rustc`, but as the number of crates to take into consideration is fixed and low, the lookup time is constant and relatively low (cf. chapter 5).

After standard item detection, the tool can turn to user code. The main procedure of the extraction enumerates all top-level items[6] of the crate. For enums and structs, the phase directly translates the HIR definition to a Stainless ADT definition. Because the enumeration order is undefined, all the top-level methods of the extraction follow the same idea: `get_or_extract`. That is, the first time an item is visited, it is extracted and its definition is stored in state, then on subsequent encounters, the definition is simply retrieved.

For functions, the extraction distinguishes between external, abstract, and local functions, but not between methods and functions because in the HIR the two are the same (with the first parameter of methods being the receiver). External and abstract functions are only extracted from the HIR as they only have a signature. For local functions, the body expression is queried from the THIR and extracted by the `expr` module which takes a `rustc_mir_build::thir::Expr` and returns a `stainless_data::ast::Expr`, i.e. it performs the translation.

Synthesis

Multiple translations in Rust-Stainless rely on the ability to synthesise expressions of certain ADT types. For it creates tuple ADTs, e.g. `Tuple3(x, y, z)`. Moreover, the fields of these ADTs also need to be available: `tuple._2`.

The synthesis module benefits from the `get_or_extract` pattern. For example, when a translation needs the option type, it triggers its synthesis. But, all synthesis methods internally implement a `get_or_create` logic, i.e. if the option type has already been either extracted from user code or synthesised by a former synthesis call, it is simply reused and the demanded ADT expression is built with the existing definition. That way, ADTs are only synthesised on demand and there is no risk of synthesising a definition that has already been extracted.

[5]For example: `stainless::Set::<T>::new`.

[6]https://doc.rust-lang.org/nightly/nightly-rustc/rustc_hir/hir/enum.ItemKind.html

```rust
trait Equals {
  fn equals(&self, x: &Self) -> bool;

  fn not_equals(&self, x: &Self)
    -> bool {
      !self.equals(x)
    }
}
trait Other<X, Y> { ... }

impl Equals for i32 { ... }


impl<T: Equals> Equals for List<T>
  { ... }

let list: List<i32> = List::Cons(
  123, Box::new(List::Nil)
);
list.equals(&List::Nil)
```

Listing 4.8: Examples of traits and implementations with and without trait bounds.

```scala
abstract class Equals[Self] {
  def equals(self: T, x: T): Boolean

  def notEquals(
    self: T, x: T
  ): Boolean =
    !this.equals(self, x)
}
abstract class Other[Self, X, Y]
  { ... }
case object i32asEquals
  extends Equals[Int] { ... }

case class ListasEquals[T](
  ev0: Equals[T]
) extends Equals[List[T]] { ... }

val list = Cons(123, Nil())
ListasEquals[i32](i32asEquals)
  .equals(list, Nil())
```

Listing 4.9: Translation as type classes (`abstract`) and implementations. Trait bounds are translated as evidence parameters.

Trait to Type Class Translation

Rust-Stainless extracts Rust traits and models them as type classes in Scala. This introduces a distinction between regular `impl` blocks for which it suffices to extract the methods as top-level functions, and `impl TraitX for TypeA` blocks that need to be extracted as type class implementations, i.e. *case classes* or *case objects*. Traits themselves are extracted as *abstract classes*, see Listing 4.8.

Internally, Rust represents `impl TraitX for TypeA` blocks with a trait bound on the block, i.e. `Self: TraitX`, while Scala uses inheritance (`extends`). Furthermore, Rust traits are implemented on the implicit `Self` type parameter, whereas Scala type classes always have at least one type parameter representing the type for which the class is implemented. Fortunately, the Rust compiler internally treats the `Self` like a regular type parameter, hence it is internally already in the Scala form.

As Listing 4.9 shows, the trait bounds on type parameters are converted to evidence parameters in Scala, like `ev0: Equals[T]`. Evidence parameters force the caller to prove that the instantiated type satisfies the bound by providing an instance of the corresponding type class. This is equivalent to Rust's compiler ensuring that `impl Equals for T` is in scope. If an implementation has no type parameter, it can be extracted as a ground case object.

If there are classes, it is also necessary to distinguish *function calls* from *method calls*. This distinction is mostly achieved by adding flags like `@abstract` and `@methodOf(i32asEquals)` to methods. The challenge of method calls is to resolve the receiver instance they are called on. For example, Listing 4.9 shows a call to `this.equals` inside the type class. In Rust, the call happens on the first parameter of the function which is also the receiver. This is implicitly resolved by the compiler. In Stainless, the call needs to happen on the type class instance and because our type class instances are only created in the extraction, the tool has to resolve the receivers itself.

Instance resolution takes a triple of type class identifier, receiver type, and type parameters, as well as the current surrounding class to resolve the receiver instance. For example, inside a type class, the `this` instance is accessible, inside classes with evidence parameters, the evidence instances are available (e.g. `ev0.equals(a, b)` in `ListasEquals[T]`) and ground case objects are always in scope. As a last resort, the instance resolution recursively checks whether it can create a new type class instance by providing it the required evidence arguments. This happens for example for the call on line 20 of Listing 4.8. It gets translated to a new instance of `ListasEquals` that is created with the ground object `i32asEquals`.

Encoding of Specifications

Listing 4.2 highlighted how specification attributes are desugared by the procedural macro of `libstainless` into an annotated closure nested inside the function. The extraction recognises the closures by their annotation and translates them to Scala's `require`, `ensures` and `decreases`. Stainless later checks that the specifications do not have any effects.

The encoding with nested closures is the third spec encoding that I explored and implemented. It unifies the advantages of the two previous designs: it works for *all* kinds of functions, be they top-level, implementation methods, or abstract trait methods; and it can use type parameters of the original function.

The first encoding desugared closures to nested functions in the original function. This served mainly to circumvent the borrow checker. The nested functions duplicated the parameters of their parent. That way, new bindings were created without borrow interference with the actual parameters but with the same types and identifiers. The problem was that this encoding was neither able to use type parameters of the surrounding function, nor the `self` parameter of methods, because neither are available to inner functions in Rust.

Therefore, the second encoding was created only for methods in `impl` blocks. Specs were desugared to sibling functions with a special name such that they could declare a `self` parameter. However, there were now two different encodings in use simultaneously and even the sibling functions couldn't provide spec attributes on trait methods. Rust does not allow additional items other than the specified methods in trait implementations.

To support specs on trait methods, the current encoding therefore uses nested closures. Closures can use surrounding type parameters, at the cost of not having a `self` parameter. To solve that, the encoding replaces `self` with a `_self: Self` parameter on the closure. Later, the extraction will again substitute this parameter with the correct receiver.

Finally, abstract trait methods do not have a body in which one can nest a closure. A simple solution is to add a body with the spec closure and an `unimplemented!` panic. Yet, that makes the method a default method which prevents the compiler from enforcing its implementation by all implementors of the trait. The solution to that last problem is conditional compilation. The inserted bodies are annotated with `#[cfg(stainless)]` which makes them vanish in normal compilation, hence, the compiler enforces that the methods are implemented, but the bodies are kept when the compiler is run with the `stainless` flag, which happens at verification.

4.4 Limitations

4.4.1 Unsupported Rust Features

In general, language features not mentioned in subsection 4.3.1 are not yet supported. However, there are mainly two features that are still mandatory to reach the goal of verifying idiomatic Rust: closures and sequences. Other missing features prevent the programmer less from writing normal code, for example, *unsafe Rust* is not supported. This is a conscious decision as many translations rely on the assumption that only *safe* Rust is used. User-annotated reference lifetimes are another example. Although, the borrow checking assumption enables the translation to ignore exact lifetimes, the tool currently does not extract explicit lifetimes. Further missing but less complex to add features are supertraits, i.e. type class inheritance in Scala, top-level constants, and Stainless invariants on ADTs.

For closures, most of the infrastructure is already in place as they are represented like functions in the compiler. There are some difficulties nonetheless. Closures can capture variables from the surrounding scope. That includes moving variables from the surrounding scope into the closure. This becomes difficult to manage for Stainless if these variables are mutable [9, section 3.4.3]. The second problem arises with higher-order functions that take closures as parameters. In Rust, the function parameter type for a closure type must be a type parameter with a trait bound like `F: Fn(i32) -> i32`, because the exact shape of the closure is not clear in advance. This polymorphism is harder to translate than directly specified lambda types, like in Scala: `Int => Int`.

By sequences, I mean vectors, arrays, iterators, but also loops. Implementing loops and even arrays should be feasible as these are already supported in Stainless [17, section “Imperative”]. On the other hand, iterators and vectors are very difficult to translate because they are not inherent *language* features but rather important and complex items of the standard library.

One Crate Limitation For standard library items, Rust-Stainless runs into the limitation imposed by the one-crate-a-time compilation model of `rustc`. The tool is currently unable to extract code of standard library items because they are not part of the user crate, hence, no HIR is available. This is currently the biggest *engineering limitation* of Rust-Stainless. Various approaches could be explored to solve it:

- If Rust-Stainless could read its binary output format and work with that, the tool could translate a crate, serialise the representation and read it again when working on the next crate. This way, the relevant items from the standard library could be extracted and later imported, when translating the user crate. The problem of that approach is that many of the widely used standard library items like vectors are implemented with very advanced features like unsafe Rust. Extracting code of that complexity is currently out of scope.

- The contrary approach would be to provide completely synthetic definitions of standard items. That is, the tool would detect items like vectors and synthesise some implementation for them, like it currently does for the Stainless set and map. While this approach would be technically feasible, it would be an unstable and labour-intensive endeavour because the synthetic shadow implementation would need to stay in sync with the standard library. Moreover, Rust’s standard library is large and it would be difficult to choose which features to support.

- The most promising way of dealing with crate-external items is to extract contracts for them that emulate the items. Indeed, this is the approach other tools take [1, 51]. The given contracts will be assumed to be correct and used to verify the user code. The library could also provide contracts for the most frequently used standard items, which is what MIRAI [51] does. Keeping such contracts up-to-date still requires less work than providing synthetic shadow implementations. The only disadvantage of this approach is that the user can state false contracts for external items, leading to incorrect results.

4.4.2 Stainless Backend Limitations

Type Classes

Traits are omnipresent in Rust. For example, Rust does not have an equality operation on all types by default, like Scala does. Rather, comparison operators are only defined in the language for primitive types. For other types, for example the `==` operator is desugared to a call to the trait method `PartialEq::eq` of the standard library. Most types then provide a derived implementation of `PartialEq` that performs structural comparison. For other operators it works the same.

On one hand, this shows how important our translation of traits to type classes is. On the other hand, it poses a problem for Stainless because structural equality exists on all types in Scala. To deal with trait methods, the frontend needs to extract the trait implementations of `PartialEq` with its type class mechanism. This either forces the user to provide an implementation for equality, or it requires the solution of the crate-external items problem, discussed in the previous section.

The second problem with type classes is that they are expensive in verification with the current Stainless pipeline. Functions that rely on type class instances are transformed by multiple phases of the Stainless pipeline, especially the *refinement lifting* [52], in such a way that verification may become untraceable in extreme cases.

A radical solution to the problem would be to erase and replace calls to `PartialEq::eq` by `==` in Scala. With a similar safety check as for the erasure of `Clone` to `freshCopy`, one can argue that derived instances of `PartialEq` can be safely replaced by the structural `==` operator of Scala.

Mutability

As subsection 3.2.1 described, the general mutability translation can be tailored around the specific aliasing restrictions of the Stainless backend. This is by itself a limitation; if Stainless worked for the entire language, one could simply use the general translation. Nonetheless, thanks to the borrow checking guarantees, the translation can be adapted rather well to the restrictions although some limitations remain, as subsection 3.2.3 discusses.

From an implementation point of view, the imperative phase of Stainless is not the most stable part of the pipeline and is also still under development. Rust-Stainless targets that phase as a new frontend in unforeseen ways and thereby found multiple bugs. Most of the bugs have already been solved.[7] However, for code using mutability in complex ways, it is still possible to find new bugs in the backend.

As discussed in the previous section, refinement lifting makes examples using type classes more expensive to verify. In combination with the mutable cell encoding and recursive types like the linked-list, this unfortunately leads two larger benchmarks of the test suite to time out.[8] Luckily, the refinement lifting phase is set to be removed from the pipeline in the future which may solve the issue.

Other Approaches To overcome the limitations of Stainless's imperative phase, a new *full imperative* phase using a *heap encoding* has been proposed [45]. That phase has explicit support for mutable references, `AnyHeapRef`, which represent parts of the heap. Furthermore, it uses annotations on functions to state which references are read (`reads`) and which are written (`modifies`) by the function. These two features are very promising to target from Rust-Stainless because they resemble things that are already in the mutability translation. `AnyHeapRef` is similar to mutable cells and it might be possible to infer the information needed for the read and write annotations from the mutability of the function parameters (which is explicit in Rust).

A radically different approach for dealing with mutability would be to extract the Rust program not from the THIR but the lower-level MIR. The next subsection is dedicated to that discussion.

4.4.3 Design Limitation

The most important design choice of Rust-Stainless is to use the THIR instead of the MIR. The THIR is well suited for translation to Stainless AST because it is still in the form of an AST but all the type information is explicit and implicit features, like method calls and dereferences, have

[7]For example an incorrect constant propagation: https://github.com/epfl-lara/stainless/issues/1090.
[8]https://github.com/epfl-lara/stainless/issues/1093

been resolved or desugared. In that sense, the THIR is the closest representation to Stainless AST and errors, counter-examples, or insights coming from Stainless are easily mapped back and forth between the two. On the downside, the THIR still has many features that all need to be understood by our translation, which becomes more complex. More importantly, the THIR does not have any information about liveness and reference lifetimes because these analyses are only performed later, on the MIR. This is the fundamental limitation of translating from the THIR.

The advantage of the MIR would be that the representation is even more explicit, there are less features to translate, and lifetimes have been resolved. That means, it would be possible to manually propagate changes back to borrowed variables at the lifetime end of mutable borrows. On the other hand, the MIR is in CFG form and it would be challenging to transform that graph back to a syntax tree for Stainless AST. It would also be nearly impossible to combine the THIR and the MIR because they do not use the same identifiers for variables.

With this chapter, the entire Rust-Stainless system has been introduced and explained. Design choices and accompanying limitations were discussed in the last section. Theoretical limitations of the mutability translation have already been discussed in subsection 3.2.3. It may be interesting to compare this system to other approaches, listed in chapter 6. The next chapter evaluates our tool under different perspectives.

5 Evaluation

This chapter presents how Rust-Stainless is tested and gives some examples of code that the tool can verify. Furthermore, it will decompose the running time of a typical execution into its subparts and finish by presenting a code example related to distributed systems implemented by Informal Systems and verified with Rust-Stainless.

5.1 Benchmarks

The Rust-Stainless repository contains a test suite of 67 passing (positive) and 12 failing (negative) test code examples of approximately 3000 lines in total.[1] The suite is run for every pull request and every commit to the `master` branch.

To quantitatively evaluate Rust-Stainless, I collected some statistics of all the positive test examples, displayed in Table 5.1. Only the positive examples were used for the measurements because the negative examples may fail at different stages of the extraction or verification pipeline. Thus, the time measurement is only representative for the positive examples which complete the verification pipeline. The used benchmarks amount to 2214 lines of Rust code (LoC), Stainless generates in total 475 verification conditions (VCs) and the accumulated running time is 5.1 minutes (Total). Note that the test suite normally (when run with `cargo test`) runs multiple tests in parallel and thus, completes in less than 2 minutes on the same machine.[2]

It is instructive to decompose the running time into its different parts. As described in subsubsection 4.3.2, the frontend first needs to detect some standard library items before it can translate the user crate. On average, detection takes 0.17 ± 0.01 s or equivalently 4 % of the running time. The actual translation is very fast, especially for the rather short examples that account for the majority of the test suite. It takes only 0.4 ms or less than 0.01 % of the running time on average. By far the largest part of the running time is not spent in the frontend itself but rather in the JVM backend. Verification time accounts for on average 95 % of the total time. The rest is made up of starting, serialising and reporting. This shows that the translation time is inferior to the verification time by multiple orders of magnitude which is not surprising, given that verification is much more complex than translation.

5.2 Code Examples

After having quantitatively measured our tool, it is time to qualitatively evaluate what it is capable of and to that end present some code examples. The first three examples stem from the test suite of Rust-Stainless and are included in the measurements from above. The fourth example is the running example from chapter 2 and the last one is an implementation by Informal Systems used in the `tendermint-rs` repository.

[1]Found under `stainless_frontend/tests/[pass|fail]` on the `mutable-cells` branch.

[2]All the tests were run on a MacBook Pro with a 2.8 GHz Quad-Core Intel Core i7 and 16 GB of RAM.

Table 5.1: Time measurements for all passing test examples of the Rust-Stainless test suite. The presented times are means and standard errors from 5 runs. The tests were performed on the `mutable-cells` branch, except for the two tests marked with * which are from `master`. All times are in milliseconds.

Name	LoC	VCs	Std Item Detection	Translation	Verification	Total
AdtUseBeforeDeclare	13	0	170.3 ± 7.9	0.3 ± 0.0	3318.5 ± 98.9	3535.3 ± 106.7
Adts	90	3	169.0 ± 3.6	0.5 ± 0.1	4142.0 ± 95.8	4361.7 ± 101.2
Blocks	27	2	177.7 ± 2.6	0.4 ± 0.1	3696.6 ± 302.0	3907.0 ± 304.2
Boxes	9	1	163.8 ± 9.5	0.3 ± 0.0	3502.3 ± 206.8	3723.5 ± 218.2
CastCorrectness	43	13	160.6 ± 3.8	0.6 ± 0.0	4683.2 ± 253.6	4926.3 ± 257.9
Clone	37	1	158.6 ± 6.3	0.4 ± 0.1	3872.3 ± 238.8	4106.7 ± 246.9
DoubleRefParam	24	5	170.6 ± 7.0	0.4 ± 0.0	3930.9 ± 146.8	4152.4 ± 150.9
ExternalFn	22	2	177.4 ± 6.1	0.4 ± 0.1	3768.7 ± 181.8	3981.8 ± 188.4
Fact	14	3	181.2 ± 5.0	0.3 ± 0.0	4415.8 ± 238.8	4633.4 ± 243.4
FnRefParam	24	5	169.8 ± 7.2	0.4 ± 0.0	3937.2 ± 189.8	4158.4 ± 198.2
GenericId	9	0	178.4 ± 3.8	0.3 ± 0.0	3238.7 ± 157.9	3441.6 ± 161.4
GenericOption	37	10	178.0 ± 3.6	0.5 ± 0.0	4130.9 ± 237.2	4339.8 ± 241.2
GenericResult	31	8	169.3 ± 8.8	0.4 ± 0.0	4108.2 ± 158.0	4333.3 ± 168.1
ImplFns	37	5	170.9 ± 6.7	0.5 ± 0.0	3973.6 ± 169.2	4205.7 ± 175.1
ImplMutSpec	19	1	176.3 ± 4.3	0.4 ± 0.0	3878.1 ± 277.6	4089.3 ± 282.4
Implies	9	1	177.4 ± 4.8	0.3 ± 0.0	3515.5 ± 142.5	3724.0 ± 147.6
InsertionSort	111	42	157.2 ± 5.7	0.9 ± 0.1	5961.8 ± 218.3	6236.6 ± 225.8
IntOperators	189	70	175.1 ± 5.7	0.9 ± 0.1	10136.4 ± 380.2	10384.3 ± 385.4
IntOption	30	1	167.4 ± 2.9	0.4 ± 0.0	3756.4 ± 118.2	3973.1 ± 121.9
LetType	19	2	157.5 ± 3.8	0.4 ± 0.1	3782.8 ± 157.2	4002.8 ± 161.9
ListBinarySearch	93	23	156.0 ± 3.7	0.8 ± 0.0	6651.8 ± 211.2	6899.7 ± 215.8
MapOps	40	31	159.2 ± 4.0	0.6 ± 0.0	5129.7 ± 129.9	5391.5 ± 134.8
Monoid	49	9	179.1 ± 5.9	0.5 ± 0.0	6637.3 ± 335.7	6849.5 ± 341.3
MutClone	11	1	155.4 ± 2.4	0.3 ± 0.0	3657.1 ± 120.1	3882.0 ± 123.3
MutLocalFields	55	6	177.2 ± 6.7	0.5 ± 0.0	4352.0 ± 224.4	4563.7 ± 230.5
MutLocalLets	29	2	156.6 ± 4.2	0.4 ± 0.0	3848.8 ± 177.4	4073.3 ± 183.0
MutLocalParams	26	3	176.7 ± 7.4	0.4 ± 0.0	3851.8 ± 145.1	4075.6 ± 152.6
MutMemReplace	60	26	156.6 ± 3.9	0.6 ± 0.1	5905.2 ± 194.0	6147.5 ± 200.6
MutOld	20	3	168.8 ± 4.6	0.4 ± 0.0	4006.1 ± 161.7	4232.7 ± 165.6
MutParams	8	0	177.2 ± 4.2	0.3 ± 0.0	3293.8 ± 82.9	3496.8 ± 87.0
MutRefBorrow0	11	2	176.7 ± 9.2	0.2 ± 0.0	3659.6 ± 151.0	3876.8 ± 160.7
MutRefBorrow1	12	1	170.1 ± 6.2	0.3 ± 0.0	3696.8 ± 127.2	3918.7 ± 135.0
MutRefBorrow10	10	3	166.0 ± 6.0	0.4 ± 0.0	3853.8 ± 175.7	4061.7 ± 181.3
MutRefBorrow11	16	3	166.3 ± 4.4	0.4 ± 0.0	3980.5 ± 113.2	4189.8 ± 118.3
MutRefBorrow12	51	27	160.2 ± 5.6	0.7 ± 0.1	5617.2 ± 159.7	5844.9 ± 166.5
MutRefBorrow2	13	1	168.7 ± 6.1	0.3 ± 0.0	3710.1 ± 141.0	3928.4 ± 147.8
MutRefBorrow3	14	1	173.5 ± 18.3	0.4 ± 0.0	3771.0 ± 219.6	3995.9 ± 238.8
MutRefBorrow5	23	3	170.1 ± 5.1	0.4 ± 0.0	3995.0 ± 171.1	4222.1 ± 175.3
MutRefBorrow6	26	0	165.8 ± 5.4	0.4 ± 0.0	3504.3 ± 129.3	3712.2 ± 134.0
MutRefBorrow7	31	3	162.6 ± 4.3	0.5 ± 0.0	4108.6 ± 98.1	4337.1 ± 104.6
MutRefBorrow8	18	3	175.6 ± 7.9	0.3 ± 0.0	3896.0 ± 122.6	4111.9 ± 131.4
MutRefBorrow9	17	3	174.9 ± 7.6	0.3 ± 0.0	4094.6 ± 544.4	4311.0 ± 552.8
MutRefClone	12	1	157.0 ± 5.2	0.3 ± 0.0	3685.4 ± 167.3	3912.6 ± 174.5
MutRefImmutBorrow	33	5	169.6 ± 6.3	0.3 ± 0.0	3955.2 ± 184.3	4175.5 ± 191.7
MutRefTuple	9	1	175.1 ± 4.8	0.3 ± 0.0	3749.6 ± 178.5	3965.2 ± 183.8
MutReturn	51	14	157.7 ± 5.3	0.6 ± 0.0	4485.3 ± 192.3	4708.7 ± 199.1
MutTuple	8	1	179.8 ± 5.5	0.3 ± 0.0	3712.1 ± 113.4	3917.6 ± 118.7
NestedSpec	16	2	179.2 ± 5.7	0.3 ± 0.0	3652.8 ± 134.4	3864.9 ± 140.1
NestedSpecImpl	20	2	172.1 ± 11.2	0.4 ± 0.0	3748.0 ± 245.4	3976.8 ± 259.6
PanicType	33	15	159.0 ± 9.0	0.5 ± 0.1	4415.7 ± 201.3	4651.8 ± 211.1
PhantomData	22	2	158.2 ± 6.2	0.4 ± 0.0	3747.4 ± 181.1	3967.0 ± 187.3
ReturnStmt	65	17	165.9 ± 6.9	0.5 ± 0.0	6032.1 ± 149.9	6269.0 ± 158.2
SetOps	20	2	158.3 ± 5.6	0.3 ± 0.0	3577.1 ± 81.1	3815.8 ± 87.5
SpecOnTraitImpl	22	3	180.2 ± 7.9	0.4 ± 0.0	4434.1 ± 172.8	4650.9 ± 180.2
Strings	39	4	164.1 ± 9.3	0.5 ± 0.0	3861.3 ± 180.7	4076.7 ± 191.5
StructUpdate	23	1	177.7 ± 7.7	0.3 ± 0.0	3766.9 ± 204.1	3974.8 ± 212.6
TraitBounds*	106	14	159.7 ± 5.8	0.7 ± 0.0	6090.8 ± 55.6	6335.7 ± 58.7
TupleMatch	12	2	174.7 ± 5.7	0.3 ± 0.1	3752.7 ± 21.8	3969.1 ± 28.7
TupleResult	9	4	162.7 ± 1.9	0.4 ± 0.1	3822.3 ± 20.4	4027.1 ± 20.3
Tuples	44	5	174.2 ± 2.9	0.5 ± 0.1	4054.1 ± 33.8	4281.5 ± 33.4
TypeClass*	83	28	157.1 ± 4.4	0.7 ± 0.0	11584.7 ± 80.2	11815.9 ± 82.0
TypeClassCallSpecs	18	2	176.5 ± 1.9	0.4 ± 0.0	3757.2 ± 9.2	3964.7 ± 10.4
TypeClassMultiLookup	59	8	166.6 ± 1.8	0.6 ± 0.0	5055.9 ± 21.8	5280.2 ± 20.5
TypeClassSpecs	28	5	177.8 ± 1.8	0.5 ± 0.1	4450.4 ± 12.1	4663.8 ± 11.2
TypeClassWithoutEvidence	27	2	166.6 ± 3.2	0.4 ± 0.0	4409.6 ± 27.3	4626.6 ± 29.5
UseLocal	19	6	177.2 ± 2.0	0.5 ± 0.0	4736.6 ± 43.2	4954.7 ± 42.1
UseStd	9	0	176.1 ± 2.7	0.2 ± 0.0	3086.8 ± 17.7	3286.5 ± 19.4

5.2.1 Test Suite Examples

Insertion Sort The first example is Listing A.2, an implementation of *insertion sort* on the already mentioned functional linked-list. The recursive data type is made possible by the support for boxes. This example was translated from an equivalent Scala example in Stainless's test suite, therefore it shows a functional way of writing Rust; there is no mutation in the entire example. In idiomatic Rust, one would probably use a vector instead of a linked-list. The example further showcases the use of implementation blocks and specs on methods. For proving termination of the recursive implementation it is crucial to add the new `measure` attribute. Also note that the `Option` type is the one from the standard library.

Because the example is completely functional, it serves perfectly to evaluate whether the mutability translation preserves functional code – despite the mutable cell encoding. To test that, consider the `sorted_insert` method, displayed in Listing 5.1, and its translation as it is submitted to Stainless, shown in Listing 5.2. The translation is almost identical. The only changes are the mutable cell patterns in the match, the `freshCopy` around the return values, and the erasure of the boxes. This is thanks to the optimisations of subsection 3.2.2. For example, the function arguments are not wrapped in cells because they're immutable.

```rust
#[pre(self.is_sorted())]
#[measure(self)]
#[post(
  ret.size() == self.size() + 1 &&
  ret.is_sorted() &&
  ret.contents().is_subset(
    &self.contents().insert(e)
  ) &&
  self.contents().insert(e).is_subset(
    &ret.contents()
  )
)]
pub fn sorted_insert(
  self,
  e: i32
)-> List<i32> {
  match self {
    List::Cons(head, tail)
      if head <= e => List::Cons(
        head,
        Box::new(tail.sorted_insert(e))
      ),
    _ => List::Cons(e, Box::new(self)),
  }
}
```

Listing 5.1: Completely functional method of the `List<i32>` in Listing A.2.

```scala
@pure def sorted_insert(
  self: List[Int],
  e: Int
): List[Int] = {
  require(is_sorted(self))
  decreases({ self })
  self match {
    case Cons(MutCell(head), MutCell(tail))
      if head <= e => freshCopy(
          Cons[Int](
            MutCell[Int](head),
            MutCell[List[Int]](
              sorted_insert(tail, e)
            )
          )
        )
    case _ => freshCopy(
        Cons[Int](
          MutCell[Int](e),
          MutCell[List[Int]](self)
        )
      )
  }
} ensuring { (ret: List[Int]) =>
  size[Int](ret) == size[Int](self) + 1 &&
  is_sorted(ret) && contents(ret).subsetOf(
    contents(self) ++ Set(e)
  ) &&
  (contents(self) ++ Set(e)) subsetOf contents(ret)
}
```

Listing 5.2: Translation of Listing 5.1.

To go a step deeper, consider the final encoding of the function, displayed in Listing A.3. Stainless submits this encoding to its solver, Inox, after having executed all its transformations. In particular, after the imperative phase has transformed all mutable fields like the one of the mutable cells to functional code. The encoding contains refinement assertions and multiple casts to use the fields of the ADT directly, e.g. `._1`, instead of using the matched bindings. Otherwise, it does not introduce unnecessary complexity to deal with mutable cells. Thus, it can be concluded that the mutability translation preserves the verifiability of functional code – at least as long as no type classes are involved that may cause trouble with the refinement lifting.

Type Class The next example in Listing A.4 is also completely immutable and demonstrates the use of a trait for equality (`Equals`) with two implementations. Note that one would usually use the traits from the standard library (`Eq` and `PartialEq`) and let the macro derive implementations for them. Here however, equality is implemented in the program because it serves as a good example for contracts on traits. The trait contains three laws corresponding to the three properties of the equivalence relation.

The implementation for `i32` is trivial except that it needs to dereference the two operands. This is to force the compiler of using the primitive comparison operator instead of `PartialEq::eq`. The linked-list implementation (Listing 5.3) is a good example of a trait bound (`T: Equals`) and two type class method calls that will be resolved in the translation.

```rust
impl<T: Equals> Equals for List<T> {
  fn equals(&self, other: &List<T>) -> bool {
    match (self, other) {
      (List::Nil, List::Nil) => true,
      (List::Cons(x, xs), List::Cons(y, ys)) => x.equals(y) && xs.equals(ys),
      _ => false,
    }
  }
  ...
}
```

Listing 5.3: Equality implemented for the linked-list.

Lastly, note that the three laws need to be reimplemented in the list implementation. This is due to Stainless not being able to recursively prove the properties without some guidance by the programmer. However, the same is necessary in an equivalent Scala example. Thus, this is not a limitation of the Rust frontend but of Stainless itself. On the other hand, the law implementations show another construct that the frontend is able to translate: static method calls like `Self::law_transitive(xs, ys, zs)` are translated to calls on the correct type class instance.

Local Mutability Turning to mutability, the benchmark in Listing 5.4 is available on the `mutable-cells` branch. It features a struct for which the macro derives a `Clone` instance. In the function, the struct is created, cloned and mutated. The assertions ensure that the cloned instance is not changed as well by the mutation. The mutability translation will replace the `clone` call with `freshCopy`.

Mutable References The running example (Listing 2.25) from chapter 2 is equivalent to the most complicated mutability benchmark on the `mutable-cells` branch[3] with the addition of a trait. In particular, it shows how a mutable reference is matched upon and then a sub-reference is returned. The sub-reference is later used to alter the container struct. In summary, the example requires support for mutable borrows, pattern matching on mutable references, passing mutable references as values, and struct mutation via references.

In the translation of `get_mut_by_id` (Listing 5.5), one can observe how the pattern match binds the mutable cell object `v` inside the tuple ADT instead of its value. Furthermore, the evidence argument responsible for the trait method, `ev0.id` on line 8, is added as parameter.

[3] https://github.com/epfl-lara/rust-stainless/blob/mutable-cells/stainless_frontend/tests/pass/mut_ref_borrow_12.rs

```
#[derive(Clone)]
pub struct S(i32);

pub fn main() {
  let mut a = S(1);
  let b = a.clone();
  a.0 = 10;
  assert!(a.0 != b.0)
}
```

Listing 5.4: Local mutation and cloning of a simple struct.

```
def get_mut_by_id[K @mutable, V @mutable](
  self: MutCell[Container[K, V]],
  id: Long,
  ev0: Id[K] @evidence
): Option[V] = {
  self.value.pair match {
    case MutCell(Some(MutCell(Tuple2(k, v))))
      if ev0.id(k.value) == id => Some[V](v)
    case _ => None[V]()
  }
} ensuring {
  (ret: Option[V]) => is_empty[K, V](self.value) ==> ret match {
    case None() => true
    case _ => false
  }
}
```

Listing 5.5: Translation of the `get_mut_by_id` method from Listing 2.25.

5.2.2 Peer-List Implementation

A principal motivation of this thesis project was to verify real-world code examples of Informal Systems. The `PeerList` data structure seemed like the perfect benchmark because it already had runtime-checked invariants defined by Romain Rüetschi. Thus, a primary driver of this thesis was to verify a version of the data structure that was as close to the original code as possible.

Thanks to the mutability translation and especially in-place updates, I achieved that goal adequately, found a problem in the original specification and was able to strengthen the postconditions of some methods. The verified version of the code is in Listing A.5 and by comparing it with the original code, one can see that there are indeed no large changes.[4]

The peer-list is a data structure that keeps track of different nodes in a distributed system context. Each node has a current value, recorded in the `values` map, and each node is either the primary node of the system, a trusted witness, a simple but working full-node, or considered as a faulty node. These states are recorded with sets in the struct and the `invariant` method returns true if the sets are in a valid configuration. For example, they all need to be disjoint as each node must be in exactly one state. The two methods of interest are `replace_faulty_witness` and `replace_faulty_primary`. Both change the state of some of the nodes.

[4]The original version can be found here: https://github.com/informalsystems/tendermint-rs/blob/d8e18c647cd8695d16610c4292b15ec6d1b45fbc/light-client/src/peer_list.rs.

The only notable but crucial change from the original code is that instead of the standard `HashMap` and `HashSet`, the verified code uses two custom implementations, `ListMap` and `ListSet`. I implemented these two specially for the benchmark. They are in a separate module in the verified crate (Listing A.6). The need for list-backed set and map implementations arose because some methods get an element from the collections with `.iter().next()` in the original code. Neither does Rust-Stainless support iterators, nor do the map and set from Stainless support element retrieval. Therefore, I created the list-backed collections and added a `first` method. Ideally, the two list-collections should be in the Stainless crate and be part of the library. If that was the case, the peer-list example would only have to import the Stainless crate and change the collection types to be verifiable. Unfortunately, the one-crate-limitation (cf. section 4.4) currently forces us to have the collections in the user crate.

Nonetheless, verifying the peer-list implementation with `cargo stainless` allowed me to improve its specifications. First, I found a problem: the precondition that the invariant must hold before the two methods was not set in the code. Furthermore, I could strengthen the postconditions of both methods to also prove that the methods perform the correct changes. For example, Listing 5.6 shows the improved postcondition of `replace_faulty_primary`. If the method succeeds and returns an `Ok` result, the `self` not only has to satisfy the invariant but the faulty primary given as argument also needs to be in the faulty nodes. This prevents the method from doing nothing which would also uphold the invariant. Note also the use of the `old(&self)` helper to use the value of `self` before the function.

```
#[post((matches!(ret, Ok(_))).implies(
  Self::invariant(&self)
    && old(&self).primary != self.primary
    && self.faulty_nodes.contains(&old(&self).primary)
    && old(&self).witnesses.contains(&self.primary)
))]
```

Listing 5.6: Postcondition of the `replace_faulty_primary` method.

The tool is also capable of verifying another example from Informal Systems. The code stems from an *inter-blockchain communication protocol (IBC)* handler implementation and was rewritten to be immutable and self-contained because it was used on an earlier version of the tool. The current implementation still verifies the example.[5]

With five different examples showcasing each their set of language features, this chapter showed what Rust-Stainless can currently process. It became clear that no matter how complicated the translations in the frontend are, the running time is dominated by the verification time of the backend. For an overview of the tool's limitations, see section 4.4. The next chapter summarises research work in related areas like ownership systems and formal verification.

[5]The PR with the example can be found here: https://github.com/informalsystems/ibc-rs/pull/759.

6 Related Work

This chapter situates the work on the translation in the context of existing research on linearly typed languages and similar approaches. It takes a closer look at some formalisations of Rust's semantics and finally compares our tool to other projects that work on verification of Rust code.

6.1 Background Topics

Rust's type and ownership system has been heavily inspired by decades of research in the programming language community. The main topics to consider are linear and unique types, ownership, as well as region-based memory management.

Linearity was introduced by Girard [19] and Wadler [53]. Linear types must be used *exactly once*, they cannot be duplicated nor discarded [2]. This presents the disadvantage that values to be reused have to be threaded through the program, e.g. a function that reads from an array also needs to return the array after the read to make it useable again. Unique types as described by Minsky [34] on the other hand guarantee that they are the only reference to a certain value, i.e. the absence of aliases. Lastly, Clarke et al. [13] introduced ownership types that follow the same goal, the absence of inadvertent and even dangerous aliasing of (heap-allocated) objects.

Many early approaches to linear and unique types originated in functional languages and some challenges arose when they were applied to object-oriented languages that may assign to values or clone objects. Shallowly copying an object in a language with owned data is problematic because it might copy unique pointers, creating illegal aliases. Deep copying would solve the problem but there is a more efficient approach called *sheep cloning*. First described by Noble et al. [37], Rust implements a version close to [29]: owned data, that is, data pointed at by unique pointers, is deeply copied, while shared data, i.e. immutable references, can be safely aliased.

To make unique types work with assignments, values need to be moved or destroyed after a read. This notion of *destructive reads* was introduced by Hogg [22], with the goal of protecting from aliasing in object-oriented languages. Rust implements such reads for its move semantics by making the borrow checker flow-sensitive, similar to the proposition of Boyland [11]. The flow-sensitive borrow checker can also be seen as substructural typing [55].

Other, less practically used languages with substructural typing or similar ideas are:

- **Mezzo**, an ML-language by Pottier and Protzenko [40], controls aliasing and ownership by offering *duplicable types* that are immutable and can be copied, and *mutable types* that are linear. This is similar to Rust's copyable and moveable types. Mezzo also uses a flow-sensitive type checker to enforce its discipline but it does that with permissions embedded in the type system rather than lifetimes.

- **Linear Haskell** by Bernardy et al. [8] tries to bring linear types to Haskell in a backwards-compatible way, i.e. existing code compiles alongside linear types. The key difference of this approach is that linearity is attached to functions instead of having linear and non-linear types like Mezzo or Rust. A linear function is restricted in the way it uses its arguments: when its result is consumed exactly once, then its argument has to be consumed exactly once.

- **Alms** is an attempt to popularise *affine types*, e.g. types that can be used *at most once*, by Tov and Pucella [48]. The language is similar to OCaml and its primary goal is to create resource-aware abstractions, for solving problems like race conditions. In an Alms module, the programmer can mark types as affine, contrary to Rust where everything is moveable, i.e. affine, unless otherwise specified (by implementing `Copy`).
- **Cyclone** was intended as a save alternative to C and is among the languages that inspired Rust the most. In particular, Grossman et al. [20] designed Cyclone with memory management based on regions. That includes ideas like region subtyping and deallocation by region. Unlike Rust, the programmer specifies whether an object should be on the heap, on the stack or in a dynamic region. In Rust, the lifetime mechanism is applied to all references and memory locations, ownership is used to trigger deallocation.

6.2 Rust Formalisations

Even if Rust is a rather young language, there have already been numerous attempts at formalising its semantics of ownership, borrowing and reference lifetimes. The following list mentions three important and recent works. Other projects are Patina [41], Rusty Types [7], and KRust [54].

- **RustBelt** by Jung et al. [24] is important because it bridges the gap of most other works: it proves the safety of implementations in *unsafe Rust* – "securing the foundations of Rust". The authors develop an automated way of proving correctness for a subset of the language and apply it to libraries that internally use unsafe features. For example, they prove that a program using the abstractions of `std::cell` runs safely if it type checks. In contrast to the following works and Rust-Stainless however, RustBelt uses a continuation-passing style language, λ_{Rust}, that is closer to the MIR than Rust itself.
- **Oxide** is a recent formalisation by some of Rust's maintainers, Weiss et al. [55]. The goal of the project is to provide formal semantics for a language close to surface Rust called Oxide. The authors present the first syntactic proof of type safety for the borrow checking of (the considered subset of) Rust and implement a type-checker for that language. The interpretation of lifetimes in Oxide is already compatible with the future version of Rust's borrow checker called *Polonius* [31].
- **Lightweight Formalism for Rust (FR)** was developed in parallel to Oxide by Pearce [38]. Like Oxide, it draws strong inspiration from Featherweight Java (FJ) [23], hence the name FR. The paper presents a calculus that models source-level Rust with its salient features like borrowing, reference lifetimes and move versus copy semantics. The author proves the soundness of the calculus but unlike Oxide, they implement the system in Java rather than in Rust. Another difference to Oxide is that FR models boxes, i.e. heap allocation. Oxide on the other hand is closer to the future version of the borrow checker and covers a larger subset of Rust.

The two latter works are of particular importance to this project because these formalisations could be used to formally prove the correctness of the translation from chapter 3.

6.3 Verification

Stainless in its current form, with the *Inox* solver [52], as developed by Hamza, Voirol, and Kunčak [21] is the successor of Leon [10]. The two systems come out of a long series of works, exploring the possibilities of verifying functional programs, mainly in Scala, with SMT solvers [5], like the early work by Suter et al. [46]. Stainless distinguishes itself from many other verifiers in that it is *counter-example complete*, that is, it produces and minimises counter-examples for all failed verification conditions.

Of particular interest to this project is the imperative phase of Stainless that was introduced by Blanc [9] and acts as main backend to the translation presented in this thesis. The new imperative phase by Schmid and Kunčak [45] is likely to overcome the current limitations of imperative code in Stainless and may be the ideal target for Rust-Stainless in the future.

Rust Verification

Due to its clear design and promising safety guarantees, Rust is predestined as implementation language for critical systems. With that arises the need to prove stronger properties on Rust programs like functional correctness. This is the major motivation behind this project, but likewise, numerous other projects try to build verifiers for Rust. The wide interest in the topic can be seen from the sheer number of projects in the following list. The selection here focusses on the projects that are closer to Rust-Stainless.[1]

- **CRUST** [47] tries to prove memory safety of Rust code using unsafe features, in the same manner as RustBelt. The approach is to translate Rust to C, generate test sequences of function calls, and then verify the code with the CBMC bounded model checker for C [26].
- **RustHorn** [33] translates Rust into *constrained Horn clauses* that can be solved by an appropriate solver. The authors prove correctness of their translation for a formalised subset of Rust including mutable references, inspired by RustBelt [24]. Contrary to Rust-Stainless, RustHorn operates on the MIR. The tool mainly proves the absence of runtime errors and has no features to express and verify specifications or higher-level properties.
- **Seer** [42] stands for *symbolic execution engine for Rust.* The engine takes the MIR and executes it symbolically with Z3 [14] as a solver backend, similar to Stainless. However, as an execution engine, the tool tries to find executions that produce errors but cannot prove higher-level properties.
- **SMACK** is a verification toolchain that translates LLVM-IR to Boogie [4] and has been extended to support Rust [3]. Like RustHorn, it is able to prove the absence of runtime errors and correctness of assertions but no higher-level properties like laws in Rust-Stainless.
- **MIRAI** [51] is an abstract interpreter for MIR, developed at Facebook. It supports very similar pre- and postconditions like Rust-Stainless that are standardised by the `contracts` crate.[2] The repository also contains contracts for many modules of the standard library. Such contracts could be a solution to Rust-Stainless's one-crate-limitation (cf. section 4.4). Contrary to Stainless, MIRAI may produce false negatives and is unable to generate counter-examples for failed verification conditions.
- **Electrolysis** [50] is probably the most similar work to Rust-Stainless. It translates Rust to the functional language Lean which is used as an interactive theorem prover [15]. Like Rust-Stainless, Electrolysis only works on the safe subset of Rust and one of the core topics of the project is the translation of mutable references. The author chooses functional lenses as representation of mutable references in Lean. This has the limitation that the translation needs to keep track of the provenance of mutable references. Unlike Rust-Stainless, this project operates on the MIR.

[1]The list of verifiers largely stems from `https://alastairreid.github.io/rust-verification-tools/`.
[2]`https://docs.rs/contracts/latest/contracts/`

- **Creusot** [16] is similar to Electrolysis, it also translates Rust to a verification language, WhyML [18]. Like MIRAI and Rust-Stainless, Creusot lets the programmer state specifications as pre- and postconditions. However, it uses a special language for the contracts, called *Pearlite*, that is developed with the project. Creusot also works with the MIR.
- **Prusti** [1] is probably the most useable tool currently, as it is the only one to offer a VSCode extension that verifies properties while the programmer is writing code. Prusti translates MIR to Viper [35], a verification framework with frontends for various mainstream languages. Like Rust-Stainless, pre- and postconditions are expressed in Rust itself. Prusti also has the ability to attach contracts to crate-external items, similar to MIRAI, solving the one-crate-limitation (cf. section 4.4). On the other hand, Prusti does not yet offer trait contract verification like Rust-Stainless with laws.

From this extensive enumeration, it is clear that Rust-Stainless is currently the only tool that operates on an IR above the MIR – the THIR. The trade-offs involved in that decision are discussed in subsection 4.4.3. Rust-Stainless could draw an advantage from that decision in the future and provide counter-example generation for Rust. That and other promising paths for the future of the tool are explored in the next and last chapter.

7 Conclusion

With this thesis, I presented Rust-Stainless, a formal verification tool for Rust based on the Stainless verifier. Building on a solid foundation of existing software and infrastructure, I added many features to the tool and thus increased its usability and expressiveness significantly. Along with smaller language features, the most important additions are immutable references and boxes, the Rust trait to Scala type class translation and, first and foremost, the mutability translation. Further, this project includes the theoretical reasoning behind the correctness of the mutability translation as well as numerous bugfixes and additions to both Rust-Stainless and Scala Stainless.

The development went hand-in-hand with testing and evaluating the tool on more and more complex Rust examples. As an intern at Informal Systems, I had access to experienced Rust developers who advised me on the choice of features to support. Chapter 5 shows how the tool was evaluated against some code of Informal Systems and it also demonstrates that the translations performed by our tool are very fast, in comparison to the time it takes to verify the code.

By combining Rust's borrow checking with Stainless's verification, Rust-Stainless gains an unimagined expressiveness. Thanks to Rust features like mutability and `mem::replace` along with Stainless's new `freshCopy` primitive, the tool can now process code that would fail Stainless's aliasing restrictions in Scala but is proven memory safe by the Rust compiler. The support for in-place updates with mutability unlocks the possibility of verifying efficient Rust code with Stainless, like a search tree – without using duplicate functional ghost structures for verification.

Extraction There are still some indispensable language features that Rust-Stainless needs to support before it can extract idiomatic Rust in all its facets. The primary ones are loops, arrays, and closures. However, the biggest limitation of the tool is the one-crate-limitation inherited from the compile model of Rust. Solving it would overcome many related problems. Instead of supporting vectors, one could add sensible contracts for vectors and iterators. More generally, with the ability to attach contracts to crate-external items, Rust-Stainless could finally use standard library items like traits for equality, ordering and hashing. Additionally, it could provide real implementations in the Stainless crate, instead of just providing bindings and replacing them with generated implementations in the extraction. This would also enable the use of verified data structures like the list-map.

Mutability The mutability translation presented in this thesis can theoretically translate all Rust mutability to Scala, but the current implementation is limited by the imperative phase of Stainless. With the current backend, this project has probably maximised the features that can be supported. Even if the imperative phase was bug-free, there are still the two drastic limitations of recursive functions returning mutable references and mutable references in mutable variables.

The promising solution to that problem could be the new fully imperative phase that is currently developed for Stainless. That phase does not have aliasing restrictions but rather its own notion of heap references. Hence, one could remove some of the Stainless specific changes to the translation, making it simpler. Additionally, I implemented the mutable cell synthesis in such a way that it would be easy to target the new heap reference cells instead.

Counter-Examples Even if Rust-Stainless seems to be the only formal verification project for Rust that translates from the THIR, instead of the lower-level MIR or LLVM-IR, this thesis project demonstrates the success of that decision. Most Rust constructs can be translated in a one-to-one fashion to Stainless AST. Even more so, Rust-Stainless could turn that decision into a virtue by becoming the first Rust verification tool that is counter-example complete! As Stainless is counter-example complete, it would suffice to translate the Scala counter-examples back to Rust.

New Applications The ability of verifying high-level properties on Rust code sparks numerous ideas for new applications of Rust-Stainless. For example, if software needs to be verified and very efficient, one can write it in Scala, verify it with Stainless and then use Stainless's C-code-generation feature. The generated code is memory safe but not as optimised as hand-written C. A much simpler approach would be to use Rust-Stainless to directly verify the efficient Rust code, given that it supports all the used language features.

Another idea is to use Rust-Stainless to verify implementations that are compiled to *Web Assembly (WASM)*. For example, the IBC relayer of Informal Systems reads policies that govern its functioning from WASM. Rust-Stainless could be used to guarantee certain properties on these policies, e.g. that no rules are in conflict with each other.

In summary, with this thesis and the implementation of the tool, Rust-Stainless, I showed that the combination of Rust and Stainless is very powerful. While the tool has its limitations, it also offers many promising paths for future work and it can already be used to verify real-world Rust code, making systems more correct and safer.

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360573

[2] H. G. Baker, ""use-once" variables and linear objects: Storage management, reflection and multi-threading," *SIGPLAN Not.*, vol. 30, no. 1, pp. 45 – 52, Jan. 1995. [Online]. Available: https://doi.org/10.1145/199818.199860

[3] M. S. Baranowski, S. He, and Z. Rakamaric, "Verifying rust programs with smack," in *ATVA*, 2018.

[4] M. Barnett, B.-Y. Chang, R. Deline, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," 09 2006, pp. 364–387.

[5] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Satisfiability modulo theories*, 1st ed., ser. Frontiers in Artificial Intelligence and Applications, 2009, vol. 185, no. 1, pp. 825–885.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177.

[7] S. Benitez, "Short paper: Rusty types for solid safety," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 69–75. [Online]. Available: https://doi.org/10.1145/2993600.2993604

[8] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear haskell: Practical linearity in a higher-order polymorphic language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158093

[9] R. Blanc, "Verification by Reduction to Functional Programs," Ph.D. dissertation, 2017.

[10] R. Blanc, V. Kunčak, E. Kneuss, and P. Suter, "An overview of the leon verification system: Verification by translation to recursive functions," in *Proceedings of the 4th Workshop on Scala*, ser. SCALA '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2489837.2489838

[11] J. Boyland, "Alias burying: Unique variables without destructive reads," *Softw. Pract. Exper.*, vol. 31, no. 6, pp. 533–553, May 2001. [Online]. Available: https://doi.org/10.1002/spe.370

[12] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[13] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," *SIGPLAN Not.*, vol. 33, no. 10, pp. 48–64, Oct. 1998. [Online]. Available: https://doi.org/10.1145/286942.286947

[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[15] L. de Moura, S. Kong, J. Avigad, F. V. Doorn, and J. von Raumer, "The lean theorem prover," 2015.

[16] X. Denis. Creusot. Accessed: 2021-07-19. [Online]. Available: https://github.com/xldenis/creusot

[17] Stainless documentation. EPFL IC LARA. Accessed: 2021-06-30. [Online]. Available: https://epfl-lara.github.io/stainless/

[18] J.-C. Filliâtre and A. Paskevich, "Why3 – Where Programs Meet Provers," in *ESOP'13 22nd European Symposium on Programming*, ser. LNCS, vol. 7792. Rome, Italy: Springer, Mar. 2013. [Online]. Available: https://hal.inria.fr/hal-00789533

[19] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," *SIGPLAN Not.*, vol. 37, no. 5, pp. 282–293, May 2002. [Online]. Available: https://doi.org/10.1145/543552.512563

[21] J. Hamza, N. Voirol, and V. Kunčak, "System FR: Formalized Foundations for the Stainless Verifiers," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360592

[22] J. Hogg, "Islands: Aliasing Protection in Object-Oriented Languages," in *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 271 – 285. [Online]. Available: https://doi.org/10.1145/117954.117975

[23] A. Igarashi, B. Pierce, and P. Wadler, "Featherweight java: A minimal core calculus for java and gj," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: Association for Computing Machinery, 1999, pp. 132–146. [Online]. Available: https://doi.org/10.1145/320384.320395

[24] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158154

[25] S. Klabnik and C. Nichols, *The Rust Programming Language.* USA: No Starch Press, 2018.

[26] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," vol. 8413, 04 2014, pp. 389–391.

[27] J. Kwon and E. Buchman, "Cosmos: A network of distributed ledgers," 2016. [Online]. Available: https://cosmos.network/whitepaper

[28] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[29] P. Li, N. Cameron, and J. Noble, "Sheep cloning with ownership types," in *Proceedings of the Workshop on Foundations of Object-Oriented Languages*, ser. FOOL, 2012.

[30] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994. [Online]. Available: https://doi.org/10.1145/197320.197383

[31] N. D. Matsakis. (2018) An alias-based formulation of the borrow checker. Accessed: 2021-07-19. [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/

[32] N. D. Matsakis and F. S. Klock, “The rust language,” in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 103–104. [Online]. Available: https://doi.org/10.1145/2663171.2663188

[33] Y. Matsushita, T. Tsukada, and N. Kobayashi, “Rusthorn: Chc-based verification for rust programs,” *Lecture Notes in Computer Science*, pp. 484–514, 2020. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-44914-8_18

[34] N. H. Minsky, “Towards alias-free pointers,” in *Proceedings of the 10th European Conference on Object-Oriented Programming*, ser. ECCOP ’96. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 189–209.

[35] P. Müller, M. Schwerhoff, and A. J. Summers, “Viper: A verification infrastructure for permission-based reasoning,” in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, ser. VMCAI 2016. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 41–62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2

[36] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[37] J. Noble, D. Clarke, and J. Potter, “Object ownership for dynamic alias protection,” 02 1999, pp. 176 – 187.

[38] D. J. Pearce, “A lightweight formalism for reference lifetimes and borrowing in rust,” *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 1, Apr. 2021. [Online]. Available: https://doi.org/10.1145/3443420

[39] ——, “A lightweight formalism for reference lifetimes and borrowing in rust,” *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 1, Apr. 2021. [Online]. Available: https://doi.org/10.1145/3443420

[40] F. Pottier and J. Protzenko, “Programming with permissions in mezzo,” *SIGPLAN Not.*, vol. 48, no. 9, pp. 173–184, Sep. 2013. [Online]. Available: https://doi.org/10.1145/2544174.2500598

[41] E. C. Reed, “Patina : A formalization of the rust programming language,” Tech. Rep., 2015.

[42] D. Renshaw. Seer: Symbolic Execution Engine for Rust. Accessed: 2021-07-19. [Online]. Available: https://github.com/dwrensha/seer

[43] Guide to Rustc Development. The Rust Programming Language. Accessed: 2021-06-28. [Online]. Available: https://rustc-dev-guide.rust-lang.org

[44] The Rust Reference. The Rust Programming Language. Accessed: 2021-06-22. [Online]. Available: https://doc.rust-lang.org/1.53.0/reference/

[45] G. S. Schmid and V. Kunčak, “Proving and disproving programs with shared mutable data,” *CoRR*, vol. abs/2103.07699, 2021. [Online]. Available: https://arxiv.org/abs/2103.07699

[46] P. Suter, A. S. Köksal, and V. Kunčak, "Satisfiability modulo recursive programs," in *Proceedings of the 18th International Conference on Static Analysis*, ser. SAS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 298–315. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041552.2041575

[47] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A Bounded Verifier for Rust (N)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 75–80.

[48] J. A. Tov and R. Pucella, "Practical affine types," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 447–458. [Online]. Available: https://doi.org/10.1145/1926385.1926436

[49] D. N. Turner, P. Wadler, and C. Mossin, "Once upon a type," in *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 1–11. [Online]. Available: https://doi.org/10.1145/224164.224168

[50] S. Ullrich, "Simple verification of rust programs via functional purification," Karlsruhe Institute of Technology (KIT), 2016.

[51] H. Venter. MIRAI: Rust mid-level IR Abstract Interpreter. Accessed: 2021-07-07. [Online]. Available: https://github.com/facebookexperimental/MIRAI

[52] N. C. Y. Voirol, "Verified functional programming," p. 229, 2019. [Online]. Available: http://infoscience.epfl.ch/record/268824

[53] P. Wadler, "Linear types can change the world!" in *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[54] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang, "Krust: A formal executable semantics of rust," in *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2018, pp. 44–51.

[55] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, "Oxide: The Essence of Rust," *CoRR*, vol. abs/1903.00982, 2019. [Online]. Available: http://arxiv.org/abs/1903.00982

[56] Wikipedia contributors. Live variable analysis — Wikipedia, the free encyclopedia. Accessed: 2021-06-22. [Online]. Available: https://en.wikipedia.org/wiki/Live_variable_analysis

[57] ——. Value (computer science) — Wikipedia, the free encyclopedia. Accessed: 2021-06-22. [Online]. Available: https://en.wikipedia.org/wiki/Value_(computer_science)#lrvalue

A Appendix

A.1 THIR Example

Listing A.1: Example of a function body in pretty-printed THIR.

```
Expr {
  ty: A,
  kind: Block {
    body: Block {
      stmts: [ Stmt {
        kind: Expr {
          expr: Expr {
            ty: [closure@examples/ex.rs:6:1: 6:20],
            kind: Closure {
              closure_id: DefId(0:10 ~ ex[b70b]::f::{closure#0}),
              substs: Closure(
                [i8, extern "rust-call" fn((A, A)) -> bool, ()],
              ),
              upvars: [],
              movability: None,
            },
          },
        },
      }],
      expr: Some(Expr {
        ty: A,
        kind: Adt {
          adt_def: A,
          variant_index: 0,
          substs: [],
          fields: [
            FieldExpr {
              name: field[0],
              expr: Expr {
                ty: i32,
                kind: Binary {
                  op: Mul,
                  lhs: Expr {
                    ty: i32,
                    kind: Field {
                      lhs: Expr {
                        ty: A,
                        kind: VarRef {
                          id: HirId {
                            owner: DefId(0:9 ~ ex[b70b]::f),
                            local_id: 2,
                          },
                        },
                      },
                    },
                    name: field[0],
                  },
                  rhs: Expr {
                    ty: i32,
                    kind: Field {
                      lhs: Expr {
                        ty: A,
                        kind: VarRef {
                          id: HirId {
                            owner: DefId(0:9 ~ ex[b70b]::f),
                            local_id: 2,
                          },
                        },
                      },
                      name: field[0],
                    },
                  },
                },
              },
            },
          ],
          base: None,
        },
      }),
    },
  },
}
```

A.2 Rust Benchmarks

These code examples are taken from the test suite of Rust-Stainless.

Insertion Sort

Listing A.2: Insertion sort, translated from the equivalent Scala benchmark of Stainless.

```rust
extern crate stainless;
use stainless::*;

pub enum List<T> {
  Nil,
  Cons(T, Box<List<T>>),
}

impl<T> List<T> {
  #[measure(self)]
  pub fn size(&self) -> u32 {
    match self {
      List::Nil => 0,
      List::Cons(_, tail) => 1 + tail.size(),
    }
  }
}

impl List<i32> {
  #[measure(self)]
  pub fn contents(&self) -> Set<i32> {
    match self {
      List::Nil => Set::new(),
      List::Cons(head, tail) => tail.contents().insert(*head),
    }
  }

  #[measure(self)]
  pub fn is_sorted(&self) -> bool {
    match self {
      List::Nil => true,
      List::Cons(x, tail) => match &**tail {
        List::Nil => true,
        // Deref integers to force primitive comparison operator
        List::Cons(y, ..) => *x <= *y && tail.is_sorted(),
      },
    }
  }

  #[measure(self)]
  pub fn min(&self) -> Option<i32> {
    match self {
      List::Nil => None,
      List::Cons(x, xs) => match xs.min() {
        None => Some(*x),
        Some(y) if *x < y => Some(*x),
        Some(y) => Some(y),
      },
    }
  }

  /// Inserting element 'e' into a sorted list 'l' produces a sorted
  /// list with the expected content and size
  #[pre(self.is_sorted())]
  #[measure(self)]
  #[post(
    ret.size() == self.size() + 1 &&
    ret.is_sorted() &&
    ret.contents().is_subset(&self.contents().insert(e)) &&
    self.contents().insert(e).is_subset(&ret.contents())
  )]
  pub fn sorted_insert(self, e: i32) -> List<i32> {
    match self {
      List::Cons(head, tail)
        if head <= e => List::Cons(head, Box::new(tail.sorted_insert(e))),
      _ => List::Cons(e, Box::new(self)),
    }
  }

  /// Insertion sort yields a sorted list of same size and content
  /// as the input list
  #[measure(self)]
  #[post(
    ret.size() == self.size() &&
    ret.is_sorted() &&
    ret.contents().is_subset(&self.contents()) &&
    self.contents().is_subset(&ret.contents())
  )]
  pub fn sort(self) -> List<i32> {
    match self {
      List::Nil => self,
      List::Cons(x, xs) => xs.sort().sorted_insert(x),
    }
  }
}

#[external]
pub fn main() {
  let list = List::Cons(
    5,
    Box::new(List::Cons(
      2,
      Box::new(List::Cons(
        4,
        Box::new(List::Cons(
          5,
          Box::new(List::Cons(
            -1,
            Box::new(List::Cons(8, Box::new(List::Nil)))
          )),
        )),
      )),
    )),
  );
  assert!(list.sort().is_sorted())
}
```

Listing A.3: Final encoding of the `sorted_insert` method from Listing A.2 in fully functional Stainless AST form that results after all Stainless transformations have passed. This is submitted to Inox.

```
def sorted_insert(self: List[Int], e: Int): List[Int] = {
  require(is_sorted(self))
  decreases(ListPrimitiveSize[Int](self))
  val t: List[Int] = {
    val t: List[Int] = self match {
      case Cons(MutCell(head), MutCell(tail)) if head <= e =>
        Cons[Int](MutCell[Int](head), MutCell[List[Int]](sorted_insert({
          val x: MutCell[List[Int]] = {
            assert({
              assert(self.isInstanceOf[Cons], "Cast␣error")
              self
            }.isInstanceOf[Cons], "Cast␣error")
            {
              assert(self.isInstanceOf[Cons], "Cast␣error")
              self
            }._1
          }
          assert(true, "Cast␣error")
          x
        }.value, e)))
      case _ =>
        Cons[Int](MutCell[Int](e), MutCell[List[Int]](self))
    }
    assert(t.isInstanceOf[Cons], "Inner␣refinement␣lifting")
    t
  }
  val res: List[Int] = {
    val res: List[Int] = t
    assert(res.isInstanceOf[Cons], "Inner␣refinement␣lifting")
    res
  }
  res
} ensuring {
  (ret: List[Int]) => {
    val t: Boolean = if (size[Int](ret) == size[Int](self) + 1) {
      is_sorted(ret)
    } else { false }
    val res: Boolean = t
    val t: Boolean = if (res) {
      contents(ret).subsetOf(contents(self) + e)
    } else { false }
    val res: Boolean = t
    val t: Boolean = if (res) {
      contents(self) + e.subsetOf(contents(ret))
    } else { false }
    val res: Boolean = t
    res
  }
}
```

Type Class

Listing A.4: Type class example for an equality trait and two implementations.

```
extern crate stainless;
use stainless::*;

pub enum List<T> {
  Nil,
  Cons(T, Box<List<T>>),
}

trait Equals {
  fn equals(&self, x: &Self) -> bool;
  fn not_equals(&self, x: &Self) -> bool {
    !self.equals(x)
  }

  #[law]
  fn law_reflexive(x: &Self) -> bool {
    x.equals(x)
  }

  #[law]
  fn law_symmetric(x: &Self, y: &Self) -> bool {
    x.equals(y) == y.equals(x)
  }

  #[law]
  fn law_transitive(x: &Self, y: &Self, z: &Self) -> bool {
    !(x.equals(y) && y.equals(z)) || x.equals(z)
  }
}

impl<T: Equals> Equals for List<T> {
  fn equals(&self, other: &List<T>) -> bool {
```

```rust
    match (self, other) {
      (List::Nil, List::Nil) => true,
      (List::Cons(x, xs), List::Cons(y, ys)) => x.equals(y) && xs.equals(ys),
      _ => false,
    }
  }

  fn law_reflexive(x: &Self) -> bool {
    match x {
      List::Cons(x, xs) => T::law_reflexive(x) && Self::law_reflexive(xs),
      List::Nil => true,
    }
  }

  fn law_symmetric(x: &Self, y: &Self) -> bool {
    match (x, y) {
      (List::Cons(x, xs), List::Cons(y, ys)) => {
        T::law_symmetric(x, y) && Self::law_symmetric(xs, ys)
      }
      _ => true,
    }
  }

  fn law_transitive(x: &Self, y: &Self, z: &Self) -> bool {
    match (x, y, z) {
      (List::Cons(x, xs), List::Cons(y, ys), List::Cons(z, zs)) => {
        T::law_transitive(x, y, z) && Self::law_transitive(xs, ys, zs)
      }
      _ => true,
    }
  }
}

impl Equals for i32 {
  fn equals(&self, y: &i32) -> bool {
    // Deref integers to force primitive comparison operator
    *self == *y
  }
}

pub fn main() {
  let a = 2;
  let b = 4;

  assert!(a.not_equals(&b));

  let list = List::Cons(123, Box::new(List::Cons(456, Box::new(List::Nil))));
  assert!(list.equals(&list));
}
```

A.3 Informal Systems Code

PeerList

The verified `PeerList` module has two files. The data structure and a supporting list-map implementation in the second one.

Listing A.5: Verified version of the PeerList data structure. The original version can be found here: https://github.com/informalsystems/tendermint-rs/blob/d8e18c647cd8695d16610c4292b15ec6d1b45fbc/light-client/src/peer_list.rs.

```rust
extern crate stainless;
use stainless::*;

mod list;
use list::*;

/// Node IDs
// PeerId was replaced by a simple u128 to make hashing easier.

pub enum ErrorKind {
    NoWitnessLeft { context: Option<Box<ErrorKind>> },
}

/// A generic container mapping ‘u128‘s to some type ‘T‘,
/// which keeps track of the primary peer, witnesses, full nodes,
/// and faulty nodes. Provides lifecycle methods to swap the primary,
/// mark witnesses as faulty, and maintains an ‘invariant‘ for
/// correctness.
#[derive(Clone)]
pub struct PeerList<T> {
    values: ListMap<u128, T>,
    primary: u128,
    witnesses: ListSet<u128>,
    full_nodes: ListSet<u128>,
    faulty_nodes: ListSet<u128>,
}

impl<T> PeerList<T> {
    /// Invariant maintained by a ‘PeerList‘
    ///
    /// ## Implements
    /// - [LCD-INV-NODES]
    pub fn invariant(peer_list: &PeerList<T>) -> bool {
        peer_list.full_nodes.is_disjoint(&peer_list.witnesses)
            && peer_list.full_nodes.is_disjoint(&peer_list.faulty_nodes)
            && peer_list.witnesses.is_disjoint(&peer_list.faulty_nodes)
            && !peer_list.witnesses.contains(&peer_list.primary)
            && !peer_list.full_nodes.contains(&peer_list.primary)
            && !peer_list.faulty_nodes.contains(&peer_list.primary)
            && peer_list.values.contains(&peer_list.primary)
            && peer_list.values.contains_all(&peer_list.witnesses)
            && peer_list.values.contains_all(&peer_list.full_nodes)
```

```
                && peer_list.values.contains_all(&peer_list.faulty_nodes)
    }

    /// Get a reference to the light client instance for the given peer id.
    pub fn get(&self, peer_id: &u128) -> Option<&T> {
        self.values.get(peer_id)
    }

    /// Get current primary peer id.
    pub fn primary_id(&self) -> u128 {
        self.primary
    }

    /// Get a reference to the current primary instance.
    pub fn primary(&self) -> &T {
        // SAFETY: Enforced by invariant
        self.values.get(&self.primary).unwrap()
    }

    /// Get all the witnesses peer ids
    pub fn witnesses_ids(&self) -> &ListSet<u128> {
        &self.witnesses
    }

    /// Get all the full nodes peer ids
    pub fn full_nodes_ids(&self) -> &ListSet<u128> {
        &self.full_nodes
    }

    /// Get all the faulty nodes peer ids
    pub fn faulty_nodes_ids(&self) -> &ListSet<u128> {
        &self.faulty_nodes
    }

    /// Remove the given peer from the list of witnesses,
    /// and mark it as faulty. Get a new witness from
    /// the list of full nodes, if there are any left.
    /// Returns the new witness, if any.
    ///
    /// ## Precondition
    /// - The given peer id must not be the primary peer id.
    /// - The given peer must be in the witness list
    #[pre(
        Self::invariant(&self)
        && !(faulty_witness == self.primary)
        && self.witnesses.contains(&faulty_witness)
    )]
    #[post(
        Self::invariant(&self)
        && !self.witnesses.contains(&faulty_witness)
        && self.faulty_nodes.contains(&faulty_witness)
```

```rust
    )]
    pub fn replace_faulty_witness(
        &mut self, faulty_witness: u128
    ) -> Option<u128> {
        let mut result = None;

        self.witnesses.remove(&faulty_witness);

        if let Some(new_witness) = self.full_nodes.first() {
            self.witnesses.insert(new_witness);
            self.full_nodes.remove(&new_witness);
            result = Some(new_witness);
        }

        self.faulty_nodes.insert(faulty_witness);
        result
    }

    /// Mark the primary as faulty and swap it for the next available
    /// witness, if any. Returns the new primary on success.
    ///
    /// ## Errors
    /// - If there are no witness left, returns ‘ErrorKind::NoWitnessLeft‘.
    #[pre(Self::invariant(&self))]
    #[post((matches!(ret, Ok(_))).implies(
        Self::invariant(&self)
            && old(&self).primary != self.primary
            && self.faulty_nodes.contains(&old(&self).primary)
            && old(&self).witnesses.contains(&self.primary)
    ))]
    pub fn replace_faulty_primary(
        &mut self,
        primary_error: Option<Box<ErrorKind>>,
    ) -> Result<u128, Box<ErrorKind>> {
        self.faulty_nodes.insert(self.primary);

        if let Some(new_primary) = self.witnesses.first() {
            self.primary = new_primary;
            self.witnesses.remove(&new_primary);
            Ok(new_primary)
        } else if let Some(err) = primary_error {
            Err(Box::new(ErrorKind::NoWitnessLeft { context: Some(err) }))
        } else {
            Err(Box::new(ErrorKind::NoWitnessLeft { context: None }))
        }
    }

    /// Get a reference to the underlying ‘HashMap‘
    pub fn values(&self) -> &ListMap<u128, T> {
        &self.values
    }
    /// Consume into the underlying `HashMap`
    pub fn into_values(self) -> ListMap<u128, T> {
        self.values
    }
}
```

Listing A.6: Supporting list-backed implementations to replace `HashMap` and `HashSet`.

```
use super::*;

#[derive(Clone)]
pub struct ListSet<T> {
    list: List<T>,
}
#[derive(Clone)]
pub struct ListMap<K, V> {
    list: List<(K, V)>,
}

#[derive(Clone)]
enum List<T> {
    Nil,
    Cons(T, Box<List<T>>),
}

impl ListSet<u128> {
    pub fn empty() -> Self {
        ListSet { list: List::Nil }
    }
    pub fn is_disjoint(&self, other: &ListSet<u128>) -> bool {
        is_equal(
            &self.list.contents().intersection(other.list.contents()),
            &Set::new(),
        )
    }
    pub fn contains(&self, t: &u128) -> bool {
        self.list.contents().contains(&t)
    }

    #[post(
      !self.contains(&t)
      && self.list.contents().is_subset(&old(&self).list.contents())
    )]
    pub fn remove(&mut self, t: &u128) {
        self.list.remove(t);
    }

    #[post(self.contains(&t))]
    pub fn insert(&mut self, t: u128) {
        self.list.insert(t);
    }
    pub fn first(&self) -> Option<u128> {
        match &self.list {
            List::Cons(t, _) => Some(*t),
            _ => None,
        }
    }
}

impl<V> ListMap<u128, V> {
    pub fn get(&self, key: &u128) -> Option<&V> {
        self.list.get(key)
    }
    pub fn contains(&self, key: &u128) -> bool {
        self.list.key_set().contains(&key)
    }
    pub fn contains_all(&self, keys: &ListSet<u128>) -> bool {
        is_equal(
            &self.list.key_set().intersection(keys.list.contents()),
            &keys.list.contents(),
        )
    }
}

fn is_equal<'a>(s1: &Set<&'a u128>, s2: &Set<&'a u128>) -> bool {
    s1.is_subset(s2) && s2.is_subset(s1)
}

impl List<u128> {
    #[measure(self)]
    pub fn contents(&self) -> Set<&u128> {
        match self {
            List::Nil => Set::new(),
            List::Cons(head, tail) => tail.contents().insert(head),
        }
    }

    #[post(
        !self.contents().contains(&t)
        && self.contents().is_subset(&old(&self).contents())
    )]
    fn remove(&mut self, t: &u128) {
        let list = std::mem::replace(self, List::Nil);
        let result = match list {
            List::Nil => List::Nil,
            List::Cons(head, mut tail) => {
                tail.remove(t);
                if head == *t {
                    *tail
                } else {
                    List::Cons(head, tail)
                }
            }
        };
        *self = result;
    }
    pub fn insert(&mut self, t: u128) {
        let list = std::mem::replace(self, List::Nil);
        *self = List::Cons(t, Box::new(list));
    }
}

impl<V> List<(u128, V)> {
    pub fn key_set(&self) -> Set<&u128> {
        match self {
            List::Nil => Set::new(),
            List::Cons(head, tail) => tail.key_set().insert(&head.0),
        }
    }
    pub fn get(&self, key: &u128) -> Option<&V> {
        match &self {
            List::Nil => None,
            List::Cons(head, _) if head.0 == *key => Some(&head.1),
            List::Cons(_, tail) => tail.get(key),
        }
    }
}
```