ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Contract Checking at Runtime and Verification-based Optimizations for a Rust Verifier

Master's Thesis

C. Hegglin

October 18, 2023

Supervisors: Aurel Bílý, Jonáš Fiala, Prof. Dr. Peter Müller

Programming Methodology Group, ETH Zürich, ETH Zürich

Abstract

Prusti is a powerful static verifier for the Rust programming language, ensuring the absence of panics and adherence to specified contracts. This thesis extends Prusti with the ability to check these contracts at runtime, through modifications to Rust's abstract syntax tree and its mid-level intermediate representation. Our implementation helps programmers to pinpoint gaps in verification or violated assumptions more easily. Additionally, this work introduces verification-based optimizations such as dead code removal and safety-check eliminations, leveraging the additional information introduced by specifications and the verification capabilities of Prusti to improve the performance of Rust programs.

Contents

Chapter 1

Introduction

Rust[6], a systems programming language, has rapidly gained popularity due to its emphasis on safety without sacrificing performance. It incorporates a unique ownership system to manage memory and concurrency, thus eliminating many classes of bugs at compile time. Prusti[1] leverages the Viper[5] verification infrastructure to annotate and verify Rust programs, enabling developers to provide formal specifications for their functions and ensure that the implementation adheres to the specified behavior.

The core goal of this thesis is twofold. Firstly, we aim to enhance Prusti with the capability to conduct runtime checks on its contracts. By integrating runtime checks into the compiled executables, we can detect violations of specifications during program execution. Secondly, we explore and implement verification-based optimizations. Harnessing the detailed information from specifications and Prusti's verification, we perform optimizations that are not always possible for a traditional compiler.

1.1 Motivation for Runtime Checks

At first glance, the integration of runtime checks with Prusti's static verification might seem redundant. If a program has been verified at compile-time, what added value can runtime checks provide? However, while static verification provides a robust layer of safety, there are situations where it may not cover all potential pitfalls. In the following sections, we will discuss the various scenarios that underline the significance of runtime checks, even in statically verified code.

Calls to unverified functions Oftentimes when verifying software, programmers need to make certain assumptions. For example, when using functionality from external crates, in Prusti programmers rely on so-called extern specifications. Giving an external function a postcondition, for example, will cause Prusti to trust this function to uphold its contract without actually verifying it. Similarly, if a function is too complex it might be infeasible to prove its specification. Prusti users will mark such methods with `#[trusted]` to skip its verification. If such a function violates its postcondition, this can render the rest of Prusti's proof invalid. Thus, a user might run into errors that are not easily linked back to the violation of the trusted contract, leading to a loss of trust in verification. Runtime checks would allow a user to detect this error early and to locate the reason for it easily.

Calls from unverified context Rust is a very popular candidate for providing libraries to other languages such as Python or Java via foreign function interfaces. Even if these functions are verified by Prusti, calls to these functions from a different language are currently not. Therefore, they can easily be called incorrectly by not satisfying their preconditions. Runtime checks detect incorrect usages and emit errors if the functions of a library are used incorrectly. The same problem occurs if we call verified functions from a trusted function.

Assume Proofs can be assisted by "assuming" additional knowledge, using the `prusti_assume` statement to infuse additional knowledge. Doing so leads the verifier to consider an expression as valid at a particular point in the code. However, if the assumed expression is not valid, this leads to an invalid proof. By checking these assumptions at runtime, developers can pinpoint and rectify erroneous assumptions.

Debugging partial specifications Annotating and verifying a program can be a challenging task. One problem in particular is the distinction between verification errors arising from incomplete specifications from those caused by incorrect specifications. Runtime checking facilitates this distinction since it can be used to "test" unverified specifications. Whenever violated specifications are detected at runtime, this confirms that our specifications must be erroneous and not only incomplete. In this context, it is beneficial to employ runtime checks for all specifications, as opposed to the specific instances we listed in previous examples.

Debugging Prusti Runtime checks could serve as a valuable tool for validating Prusti's correctness. Specifically, if a Rust program successfully undergoes Prusti's verification but later experiences contract violations at runtime, this indicates a potential unsoundness within Prusti itself.

1.2 Overview

In this report, we will first explain necessary background information concerning Rust, the Rust compiler, and Prusti in Chapter 2. Chapter 3 revolves around the manual rewriting of Rust programs with Prusti specifications for runtime checks, and employing static verification to detect unreachable code. Chapter 4 contains the details of our implementation. In particular, we discuss how runtime checks can be automatically translated and inserted into the generated executables by interacting with the Rust compiler and its internal data structures. Additionally, we explain how similar modifications are performed to achieve the mentioned optimizations. We qualitatively evaluate and discuss the results of our implementation in Chapter 5, before concluding our work and discussing opportunities for future work in Chapter 6.

Chapter 2

Background

2.1 Rust

While readers are expected to have a foundational understanding of Rust, this section elaborates on topics vital for comprehending the contents of this thesis. We will also explore lower-level details that might not be well-known even to seasoned Rustaceans.

2.1.1 Ownership

In Rust, the concept of ownership plays a pivotal role in ensuring memory safety without requiring a garbage collector. At its core, ownership establishes that each value in the program has a single designated owner, which is a variable. When the owner goes out of scope, its associated memory is automatically reclaimed.

Move Semantics

When a value is assigned from one variable to another, or passed as a function argument, Rust by default "moves" the value, transferring its ownership. After the move, the original variable becomes inaccessible for further use, ensuring that there is only one owner at any given time.

For instance, in Listing 2.1 the variable `x` is moved into the function `consume` and can no longer be accessed afterward, causing the program to fail compilation. Since the value has been moved into a function, this also means that this function is responsible for freeing the memory of the moved vector at the end of its execution.

Listing 2.1 Example of trying to access a value that is no longer owned.

```
fn main() {
    let x = vec![1,2,3];
    consume(x);
    let y = x.get(1); // illegal, x is no longer valid
}

fn consume(v: Vec<i32>) {}
```

Listing 2.2 Example call to a function where the passed argument is not consumed because it implements `Copy`.

```
fn main() {
    let x = 5;
    non_consuming(x);
    let y = x; // x is still accessible
}

fn non_consuming(v: i32) {}
```

Copy and Clone

Rust's `Copy` trait provides a mechanism to sidestep the move semantics, allowing certain types to be duplicated merely by copying their bits. Whenever a variable of a `Copy` type is assigned to another or passed as a function argument, the data is duplicated and both the original and the duplicate can be used independently. This is a shallow bit-wise duplication, and this trait cannot be safely implemented for heap-allocated types.

An example is shown in Listing 2.2, where the difference to our previous example is the fact that the passed type `i32` implements `Copy`, making this a valid program.

The `Clone` trait, on the other hand, offers a way to explicitly create a duplicate of an object. The trait is more versatile and is appropriate for types that require a deeper form of copying, and thus suitable for types managing heap-allocated data. For instance, it can be used to fix the error of the earlier example in Listing 2.1, by calling `consume(x.clone())` instead of moving `x` into the function. The `.clone()` method must be explicitly used to create a duplicate. Implementations of `Clone` for heap-allocated data necessarily involve allocating additional memory for the duplicates. Notably, all `Copy` types also implement `Clone`. The reverse, however, is not true.

2.1.2 Borrowing

Borrowing in Rust permits sharing access to data in a safe manner while respecting the rules of the ownership system. A borrow is either mutable or immutable.

Listing 2.3 Multiple immutable borrows of the same value.

```rust
let s = String::from("hello");
let r1 = &s;
let r2 = &s;
// both r1 and r2 can be used simultaneously
```

Listing 2.4 A mutable borrow followed by another illegal immutable borrow of the same value.

```rust
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &s; // illegal since r1 is still accessed later
r1.push_str(" world");
```

Immutable Borrowing

Immutable borrowing allows multiple parts of the code to read from the same data simultaneously without being able to modify it. When data is immutably borrowed, no mutable references to that data are allowed to coexist with immutable references. This ensures that the data will not change unexpectedly while being read. An example is shown in Listing 2.3.

However, Rust offers a concept known as interior mutability which allows for mutation through an immutable reference. This is facilitated by types like `RefCell` and `Mutex` which employ runtime checks to ensure safety.

Mutable Borrowing

Mutable borrowing, on the other hand, grants exclusive access to data for reading and writing. Only one mutable reference to a particular piece of data is allowed at any given time. Trying to borrow a value that is already mutably borrowed leads to an error at compile time. This exclusivity prevents data races, which are a common source of concurrency bugs in many programming languages. An example of an illegal sequence of borrows is shown in Listing 2.4.

Borrow Checking

The borrow checker is an integral component of the Rust compiler, tasked with ensuring that references strictly adhere to Rust's borrowing and ownership rules at compile time. The primary goal of the borrow checker is to ensure that references do not outlive the data they point to and that mutable and immutable references cannot coexist simultaneously.

Polonius[2] is an implementation of the borrow checker that seeks to refine and extend Rust's borrow-checking capabilities. The results of its analysis are used by Prusti. Polonius introduces the notion of "loans", representing

Listing 2.5

```
1  struct Percentage(usize)
2
3  fn main() {
4      let mut p = Percentage(42);
5      let x = {
6          let y = &mut p; // ($L_1$)
7          &mut p.0        // ($L_1$, $L_2$)
8      }
9      *x = 56;
10     // $L_2$ expires, then $L_1$ expires
11     p.0 = 72;
12 }
```

the code span during which a particular piece of data is borrowed via a reference. The state of a loan is analyzed across positions of a program, marking it as *alive* after its creation until it becomes *dead* at another position. The transition point is termed the loan's *expiration location*, marking the last location where the contents of a borrow could be legally accessed.

Note that the life cycle of a loan does not necessarily align with the scope of its corresponding reference. To illustrate, consider the example shown in Listing 2.5. Here, the first loan labeled as L_1 is initiated on line 6. This reference handed out to y is subsequently reborrowed, creating the loan L_2. Although the variable y goes out of scope, the loan L_1 has to be alive for at least the duration of L_2.

The borrow checking rules would permit us to access the contents of y on line 10, right after L_2 expires and before L_1 expires itself. Line 11 requires both loans to be dead, for p to be accessible again.

2.1.3 Rust Macros

Rust macros are a metaprogramming tool that allows for code generation and transformation. This mechanism operates at the syntactic level before the actual compilation process begins.

There are two primary types of macros in Rust: declarative macros and procedural macros. Declarative macros are defined by a set of rules that describe how the input tokens should be transformed to generate code. Contrastingly, procedural macros are expressed using actual Rust code, allowing for a more unrestricted analysis and modification of the passed tokens. For the purposes of this thesis, we will discuss procedural macros due to their extensive application in our work.

Beyond the differences between procedural and declarative macros, it is essential to also understand the distinction between function-like macros and attribute macros. Examples of declarations and usages of both function-like

Listing 2.6 Example of function-like macros versus attribute macros.

```rust
// defines a procedural function-like macro
#[proc_macro]
fn function_like(tokens: TokenStream) -> TokenStream {..}

// defines a procedural attribute macro
#[proc_macro_attr]
fn attr_macro(attr: TokenStream, tokens: TokenStream) -> TokenStream
↪  {..}

// uses attribute macro on function foo
#[attr_macro(args)]
fn foo() {
    // uses function-like macro
    function_like!(args);
}
```

and attribute macros are shown in Listing 2.6. Function-like macros accept a sequence of tokens and produce a new token stream as output. Attribute macros, on the other hand, can be attached to items like functions or structs and are able to modify them. When processed they have access to both the argument that was passed to them explicitly, as well as the tokens of the item they are attached to (the tokens of the function `foo` in the previous example). When these macros are expanded, the declared functions are executed to produce a modified version of the original items or even entirely new items.

2.1.4 The Rust Compiler

A large part of this thesis is interacting with the Rust compiler and altering it to instrument the generated executables. Explanations require a good understanding of the process of compiling a Rust program and its internal data structures. In this section, we will talk about the various phases the Rust compiler goes through and some of its intermediate representations.

Compiler Pipeline

Initially, raw Rust code is lexed and parsed into an AST. This representation is further modified during Macro Expansion, producing an altered version of the AST. As the compiler progresses, it transforms the AST into the High-Level Intermediate Representation (HIR), representing the AST in a more desugared and resolved form and used to perform type checking. The typed HIR (THIR) is then lowered to the Mid-Level Intermediate Representation (MIR), a control-flow graph representation of the program. The MIR is the central representation of the Rust compiler used for many analyses and optimizations of the program. Upon its initial construction in a representation

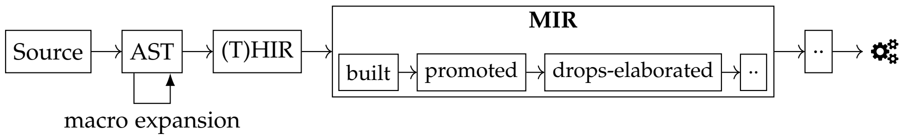


Figure 2.1: A simplified illustration of the Rust compiler pipeline.

referred to as `mir_built`, the MIR undergoes several transformation and optimization phases resulting in different versions of the MIR.

Once the MIR phase concludes, the code undergoes backend compilation to be further optimized and transformed into machine code for the targeted architecture. An illustration of this pipeline is shown in Figure 2.1.

The MIR

The MIR is of central importance to this thesis, since both Prusti in its original state and our implementation heavily rely on it. Some of its key characteristics are that it has no nested expressions and all its types are fully explicit. In the MIR, a program is represented as a collection of items, each representing an entity like a function, a trait, or a type. Each item has a unique identifier referred to as its `DefId`.

Each function item of the MIR is represented by a struct called `Body`, containing its control-flow graph, function signature, a list of declared variables, and more. In the following, we explain the terminology used for the MIR and take a closer look at its structure.

Locals, Places, and RValues *Locals* refer to memory locations on the stack such as function arguments, local variables, and temporaries. They are indexed and written with a leading underscore, e.g. `_1`. For a function with *n* parameters, the locals `_1` to `_n` are always its parameters and `_0` is its return value.

Places are expressions that identify a location in memory. A place is always constructed from a local and a set of projections. The projections are used to model things such as field accesses, indexing, or subslicing.

Examples: `_1`, `_1.f`, `_1[2]`

Operands in Rust represent values and can originate from either loading a memory location, referred to as a place, or directly from a constant value. Operands are often used as the input to various MIR instructions, serving as the means by which data flows through the computation.

RValues stand for "right-hand values" and represent the values that can be assigned to a place in the MIR. RValues encompass a broad range of computational expressions, not only including simple values but also binary operations (like addition or multiplication), unary operations (like negation), and other computations. An assignment statement in the MIR consists of a place as its destination and the source of the assignment which is an RValue.

Control-flow Graph Each node in the CFG corresponds to a basic block consisting of a sequence of statements and a terminator. Basic blocks are indexed starting at 0, which is always the entry block for a function. Oftentimes we refer to `locations` in the MIR, which point to a specific statement or terminator. Locations consist of a block index and a statement index, usually denoted as `bbx[y]` to point to the statement `y` in block `x`. Consider the simple example function shown in Listing 2.7 and its corresponding CFG in Figure 2.2. We will use this example to discuss the structure of the MIR in more detail.

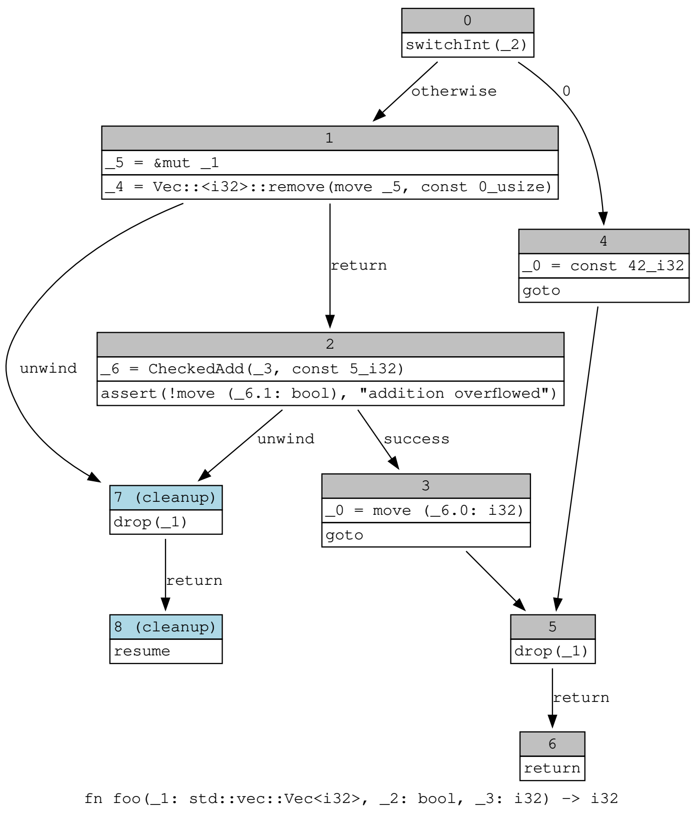
Terminators There are multiple kinds of terminators in the MIR. The ones that are relevant to this thesis will be discussed in the following.

- The **Return** terminator marks the successful termination of a function. An example is shown in block 5 of our example.
- The **Resume** terminator indicates that the function has encountered a panic and consequently terminates its execution, signaling this event to the caller.
- **Call** terminators invoke a function with a sequence of places as its arguments. The result of the call is assigned to its destination. A call terminator has two outgoing edges called `return` and `unwind`. The path taken at runtime depends on whether the called function panicked or returned. If the called function panicked, the caller proceeds on the unwind path, usually cleaning up any heap-allocated data it owns and subsequently resuming itself, leading to the unwinding of the full call stack. We say "usually" because it is technically possible to "catch" panics, but it is usually discouraged in idiomatic Rust. Otherwise, the execution continues on the return path. An example of a call is shown in block 1 of our example.
- **Assert** terminators have one Boolean place as an argument. If that Boolean is true, the execution jumps to its `success` target and otherwise the function panics as well. Assert terminators are not generated for user assertions, rather they are a result of Rust's runtime checks for certain critical operations. For example, they are used to check that certain binary operations do not overflow or to check that an index of a slice is within bounds.

Listing 2.7 A simple Rust function.

```rust
fn foo(mut v: Vec<i32>, b: bool, x: i32) -> i32 {
    if b {
        v.remove(0);
        x + 5
    } else {
        42
    }
}
```

Figure 2.2: MIR control flow graph generated for the function in Listing 2.7.


0
switchInt(_2)
otherwise
0
1
_5 = &mut _1
_4 = Vec::<i32>::remove(move _5, const 0_usize)
return
4
_0 = const 42_i32
goto
unwind
2
_6 = CheckedAdd(_3, const 5_i32)
assert(!move (_6.1: bool), "addition overflowed")
unwind
success
7 (cleanup)
drop(_1)
3
_0 = move (_6.0: i32)
goto
return
8 (cleanup)
resume
5
drop(_1)
return
6
return
fn foo(_1: std::vec::Vec<i32>, _2: bool, _3: i32) -> i32

- The **Goto** terminator represents an unconditional jump, directing the flow to its designated target.
- **SwitchInt** is used to represent conditional branching. It is the result of if-else or match statements for example. The terminator takes one integer place as an argument, which is referred to as its discriminant. It jumps to a different block according to a jumplist, or the target labeled `otherwise` if no entry in this list matches its discriminant. An example with a jumplist of length one is shown in block 0.
- The **Drop** terminator invokes the de-allocation of a value. Every heap-allocated value owned by the function must be dropped before the function terminates. In our current example, the vector in the function parameter `v` is dropped in block 5 of the MIR.

Compiler Queries

The Rust compiler has to answer numerous questions about the code during compilation, such as the MIR representation of a particular item at a specific stage of its compilation. Instead of computing these details from scratch every time they are requested, the compiler uses a demand-driven system called "compiler queries". The results of these queries are cached so the compiler can reuse them. Recalling our compiler diagram shown in Figure 2.1, each of the MIR versions corresponds to one compiler query.

However, to avoid duplicating these potentially large data structures every time they are transformed, the compiler also uses a principle called "stealing". For example, when `mir_built` is constructed for an item, it is then cached and stored. At a later point, when the `mir_promoted` query is invoked, it will *steal* the `mir_built` representation of that item and start modifying it in place. As a consequence, the `mir_built` query can no longer be invoked from this point onwards.

Compiler Interface

The Rust compiler interface serves as the main point of contact for external tools such as Prusti, interfacing with the Rust compiler. It provides a set of tools and libraries to interact with and leverage the Rust compiler's capabilities. It allows applications to hook into specific stages of the compilation process via callbacks to analyze or modify internal data structures. Unfortunately, the interface is highly unstable, and applications relying on it will require changes with each update of the Rust compiler.

Applications using this interface can use compiler queries to obtain information about the program that is being compiled. Queries can also be overridden, giving us the option to modify their results.

Listing 2.8 Function with a pre- and postcondition.

```
#[requires(x > 0)]
#[ensures(result < 0)]
fn invert(x: i32) -> i32 { -x }
```

2.2 Prusti

In this section, we will give an overview of Prusti's contracts, its specification language, and some details of its inner workings.

2.2.1 Prusti Specifications

Prusti allows programmers to extend their Rust programs with various annotations to prove certain properties.

Specification Items

Pre- and Postconditions Functions can be annotated with pre- and postconditions as shown in Listing 2.8, expressing Boolean conditions that need to be satisfied upon entry or exit of the function, respectively.

Pledges Prusti's concept of pledges is a powerful tool to reason about mutably borrowed values over the span of their life. This abstraction ensures that despite potential modifications, certain properties about these values remain consistent. The introduction of pledges in Prusti was motivated by the need for a more refined mechanism to assure properties about borrowed values beyond their immediate use.

Prusti contains two kinds of constructs to express pledges. The first kind of pledge is `assert_on_expiry`, which takes two Boolean expressions as arguments. We will refer to them as the left-hand side and right-hand side of the pledge. The left-hand side of a pledge is used to reason about a mutable reference that was handed out by a function and its state when it expires. The right-hand side is used to express properties of the value that was borrowed from, that need to be re-established once the reference expires. The pledge does not restrict intermediate mutations while the borrow is alive; rather, it emphasizes the state of the value at the moment of expiration.

Consider the example shown in Listing 2.9 containing a method `get_mut` that hands out a mutable reference to the field of the struct `Percentage`. The pledge attached to this method states that the values assigned to the contents of this reference remain between zero and 100. In the main function, we see an example of the pledge in action. After obtaining a mutable reference to the `Percentage` struct's internal value using `get_mut`, its value can be modified as shown on line 16. The loan that was handed out expires on line

Listing 2.9

```
1  struct Percentage(usize);
2
3  impl Percentage{
4      #[assert_on_expiry(
5          *result <= 100,
6          before_expiry(*result) == self.0 && self.0 <= 100
7      )]
8      fn get_mut(&mut self) -> &mut usize {
9          &mut self.0
10     }
11 }
12
13 fn main() {
14     let p = Percentage(42);
15     let x = get_mut(&mut p);
16     *x = 72;
17     // x expires
18     p.x = 101;
19 }
20
```

17, which is where the conditions expressed by the pledge must hold, which is the case in this example. Had we assigned a value greater than 100 to `*x`, this program would fail verification.

The second construct considered a pledge is `after_expiry`. It can be viewed as a simplified version of `assert_on_expiry` since a specification of the form `#[after_expiry(expr)]` is equivalent to `#[assert_on_expiry(true, expr)]`.

Inline Assertions With the use of constructs such as `prusti_assert!()`, developers can assert expected conditions at particular points in the code. Similarly, `prusti_assume!()` allows developers to make assumptions that Prusti takes for granted during verification. The `body_invariant!()` is yet another powerful construct, employed within loops to denote invariants — conditions that must remain true for every iteration. Note that by default all of these specifications are solely used for verification purposes and have no influence on the behavior of the program at runtime.

Specification Language

Prusti's specification language extends Rust's native syntax by allowing a variety of specialized constructs to express more intricate properties and behaviors. While contracts typically employ Boolean expressions, Prusti introduces additional features for expressive program verification:

- **old**: This construct captures the value of an expression at the beginning of a function, facilitating comparison with its value at the function's termination.
- **quantifiers**: For more generalized conditions, Prusti introduces universal (`forall`) and existential (`exists`) quantifiers.
- **result**: In postconditions or pledges, the result keyword represents the return value of the function. This provides a way to specify conditions on what a function should return.
- **syntactic sugar**: Prusti introduces syntactic sugar for multiple logical expressions and comparisons. Examples include the operators ==> for logical implication or === for snapshot equality. These operators offer a clearer, more concise way to express properties.

Contract Rewriting

As the reader might have noticed, all specification items are either attribute macros or function-like macros. When Prusti is executed for a given program, all contract macros are expanded to produce a modified program. For the context of our thesis, a few aspects of this rewriting are very substantial.

Preparsing Expressions within contracts are processed by a custom parser, that desugars syntactic sugar, like implications, and translates them to valid Rust expressions. For example an implication of the form `a ==> b`, which is not valid Rust syntax, is rewritten to `!a || b`.

Generation of Specification Functions Contracts attached to functions will be translated to so-called specification functions. This enables type checking of contracts and allows Prusti to access the lower-level representations of these functions to encode specifications. For instance, consider the following example program containing a function with a precondition:

```
#[requires(a % 2 == 0)]
fn foo(a: i32) {}
```

After macro expansion has taken place, this program will be rewritten to:[1]

```
#[prusti::spec_only]
#[prusti::spec_id = "dc7f47"]
fn prusti_pre_item_foo_dc7f47(a: i32) -> bool {
    let prusti_result: bool = a % 2 == 0;
    prusti_result
}
#[prusti::pre_spec_id_ref = "dc7f47"]
fn foo(a: i32) {}
```

[1]UUIDs in this example have been shortened for the sake of presentation.

In this transformed code, every specification function is uniquely identified. The attribute `#[spec_only]` indicates to Prusti that the function is a specification function and should not undergo the regular verification process. By adding `spec_id` attributes to both the original function and the generated specification function, a link between the original code and its associated specification is constructed. This connection ensures that, during Prusti's verification of the initial function, it can directly access and use the associated specification.

Inline Assertions Inline assertions in Prusti are treated separately because they are processed as function-like macros. For instance, the assertion:

```
prusti_assert!(x % 2 == 0);
```

is rewritten to:

```
if false {
    #[prusti::spec_only]
    #[prusti::prusti_assertion]
    #[prusti::spec_id = "b1d6a0"]
    || -> bool { x % 2 == 0 };
};
```

This transformation places the expression inside a closure. The reasoning for this choice is that, at the AST level, no type information regarding the operands used can be obtained. Consequently, creating a specification function would be infeasible. By utilizing a closure, the variables in scope can be effectively captured. Again, the `#[spec_only]` attribute enables Prusti to distinguish code generated from specifications from code written by the user.

2.2.2 Interaction with the Rust Compiler Interface

To verify Rust programs, Prusti makes use of the previously mentioned Rust compiler interface.

Running Prusti results in the Rust compiler being invoked, and Prusti performs its verification during the compilation process. To be precise, it uses the `mir_promoted` phase of the MIR and encodes it to the Viper verification language. This phase was chosen for multiple reasons. First of all, it still contains certain debug information that links MIR primitives to locations in the source code. Secondly, this is the phase that is used to perform the borrow checking. Therefore, information generated by the borrow checker such as expiration locations is only accurate during this phase.

After encoding the MIR to a Viper program, this program is verified. If errors are generated they can be linked back to locations in the MIR, and subsequently back to locations in the original Rust program.

Chapter 3

Approach

3.1 Translation

In this chapter, we will first have a look at how Prusti's specifications can be checked at runtime according to their semantics. We focus on how programs that are annotated with contracts can be rewritten by extending them with assertions, to ensure the annotated properties are checked correctly at runtime.

In Section 3.1.1 we look at the different kinds of contracts (such as preconditions and postconditions) and where their corresponding checks should be performed. Subsequently in Section 3.1.2 we discuss how certain Prusti-specific contents of these contracts are translated. How exactly these contracts are *automatically* translated and inserted into executables will be discussed in Chapter 4.

Remark 3.1 *Although the translation of Prusti-specific expressions contained in contracts is discussed in the second part of this chapter, we already utilize it in preceding sections. We denote their translation as the function $T : Expr \rightarrow Expr$. When this expression is encountered within a code snippet, it does not represent a function being invoked. Rather, it signifies that this expression still has to be rewritten to account for any occurrences of Prusti features that are not evaluated correctly at runtime.*

3.1.1 Specification Items

Preconditions

Prusti enables users to annotate functions with preconditions, denoting conditions that must be satisfied upon the function entry. One approach to check these preconditions is to modify the body of the function directly by inserting assertions at the beginning of the function. An example of this translation is shown in Listing 3.1.

Listing 3.1 Insertion of a precondition check into the function's body.

```
#[requires(expr)]
fn foo(input: I) {
    assert!(T(expr));
    // rest of function body
}
```

Listing 3.2 Insertion of a postcondition check into the function's body.

```
#[ensures(expr)]
fn bar(input: I) -> S {
    if b {
        assert!(T(expr));
        return res1
    }
    assert!(T(expr));
    res2
}
```

However, if this is a precondition of an external function, for example, a function of the standard library, we cannot modify its body. In such instances, all calls to this function within our program are prepended with a check instead.

Postconditions

Similarly, postconditions are used to specify conditions that need to be satisfied after the execution of a function. Checking a postcondition by altering the function's body, is slightly more intricate in this scenario. Every location that returns from the function needs to be prepended with a runtime check. An illustrative example is shown in Listing 3.2.

When the function body cannot be modified, we instead append runtime checks to each invocation of functions with a postcondition.

Pledges

Unlike preconditions and postconditions, pledges cannot be checked by modifying the body of the function they are attached to. They describe conditions that have to be satisfied after the method has returned, necessitating checks at the call site. To insert a check, we must examine the reference returned by the call and determine where it expires. Technically, the left-hand side of a pledge needs to be checked right before a loan expires, accessing the reference that was handed out at the last point where it still can be accessed. The right-hand side is checked as soon as the original value which was borrowed from becomes accessible again. This implies that it needs to be checked after the expiration of the loan. Given that loans always ex-

Listing 3.3 Insertion of runtime checks for a pledge.

```
#[assert_on_expiry(*result <= 100, p.0 <= 100)]
fn get_mut(p: &mut Percentage) -> &mut usize {
    &mut p.0
}

fn main() {
    let a = Percentage(42);
    let x = a.get_mut();
    *x = 67;
    assert!(*x <= 100);
    // a expires
    assert!(a.0 <= 100);
}
```

pire between statements, we only need to ensure that the left-hand side is checked before the right-hand side to preserve the order described above. A simple example is shown in Listing 3.3.

While this would be the correct way of translating the previous example, the method of translating pledges as described so far is not sufficient in all cases. For an example, refer to Listing 3.4. In this example, there are two kinds of pledges with different restrictions on the same type. Variable `r` holds a reference that must satisfy one of the two pledges when it expires. Which of the two pledges needs to be checked depends on the path taken at runtime.

To encode this scenario correctly, guards are added for each encountered pledge. When a borrowing function with an associated pledge is called, the corresponding guard is set to true. The check at the expiration locations is only performed if the guard related to the pledge has been set to true. Additionally, to denote the pledge as expired post-check, the guard is reset to false. This is required when a reference holding a pledge is conditionally overwritten with a different pledge. The translation of our previous example is shown in Listing 3.5.

Inline Assertions

In Prusti, several constructs can be embedded within any code block, in contrast to being attached to elements like functions. Specifically, we are referring to `prusti_assert`, `prusti_assume` and `body_invariant`. All of them use Prusti specification syntax to express that certain properties hold at specific locations of a program. In Prusti's original state, compiling a program containing such statements has no impact on the program's runtime behavior. Although the verifier interprets each of the three statements differently, for the purpose of runtime checking we handle them all in the same way.

Listing 3.4 Example of a reference that contains one of two pledges depending on the path taken at runtime.

```
#[assert_on_expiry(
    result >= 50 && result <= 100,
    p.0 >= 50 && p.0 <= 100
)]
fn get_likely(p: &mut Percentage) -> &mut usize {
    &mut p.0
}

#[assert_on_expiry(result <= 50, p.0 <= 50)]
fn get_unlikely(p: &mut Percentage) -> &mut usize {
    &mut p.0
}

fn bar(b: bool, x: usize) {
    let p = Percentage(50);
    let r = if b {
        get_likely(&mut p)
    } else {
        get_unlikely(&mut p)
    }
    *r = x;
    // End of loan
}
```

Listing 3.5 Usage of the pledges defined in Listing 3.4 and added modifications to check them at runtime.

```
fn bar(b: bool, x: usize) {
    let mut guard_1 = false;
    let mut guard_2 = false;
    let p = Percentage(50);
    let r = if b {
        guard_1 = true;
        get_likely(&mut p)
    } else {
        guard_2 = true;
        get_unlikely(&mut p)
    }
    *r = x;
    // End of loan
    if guard_1 {
        assert!(*r >= 50 && *r <= 100);
        assert!(p.0 >= 50 && p.0 <= 100);
        guard_1 = false;
    }
    if guard_2 {
        assert!(*r <= 50);
        assert!(p.0 <= 50);
        guard_2 = false;
    }
}
```

Listing 3.6 Translation of inline assertions.

```
fn main() {                           fn main() {
    prusti_assume!(expr1);                assert!(T(expr1));
    prusti_assert!(expr2);                assert!(T(expr2));
    for i in 0..10 {                      for i in 0..10 {
        body_invariant!(expr3);               assert!(T(expr3));
    }                                     }
}                                     }
```

It is worth noting that traditional loop invariants must also be met when exiting the loop, whereas the semantics of `body_invariant` do not require the condition to be true on exit. Otherwise, the encoding of body invariants would differ from that of assertions and assumptions.

To check inline assertions at runtime, we only need to translate the expressions they contain and put them into "real" assertions, as illustrated in Listing 3.6.

Predicates

Predicates in Prusti are used to make specifications more modular and abstract over certain implementation details. If an abstract predicate[1] is used within a specification we cannot check it at runtime. Predicates with an implementation can be used, but their contents will also need to be translated.

3.1.2 Specification Language

To check contracts at runtime, the expressions within contracts need to be translated according to their semantics, which is why we introduced the rewriting function T earlier. In the following, we will examine several features of the Prusti specification language, and discuss how they can be translated to ensure accurate behavior at runtime.

Result and Variable Names

In postconditions and pledges, Prusti specifications refer to the returned result of a function by using the `result` keyword. We must ensure that the result of a function is accessible, thus it needs to be assigned to a variable. We then simply replace every occurrence of `result` in the specification with the name of the correct variable. When checking contracts on the call site of a function, we also need to ensure that the names of parameters in the specification are substituted with the names of the arguments that were passed to it.

[1]A predicate without a body

Listing 3.7 Primitive encoding of old expressions.

```
#[ensures(*x == old(*x) + 1)]
fn increment(x: &mut i32) {
    x + 1
}
```

```
let old_x = *x;
increment(&mut x);
assert!(*x == old_x + 1)
```

Listing 3.8 Example of an old expression that is only conditionally evaluated.

```
#[ensures(index < old(v.len()) ==> old(v[index]) == result)]
fn remove(v: Vec<i32>, index: usize) -> Option<usize> {..}
```

Old

The most intuitive way to evaluate `old` expressions at runtime would be to evaluate and store each expression that occurs within an old expression at the correct location. For instance, for calls to the function shown in Listing 3.7 we would have to store the old value of `*x` prior to invoking the function. This stored value would then be used to check the postcondition.

Unfortunately, there are multiple reasons why this encoding is not sufficient. Firstly, storing a value by dereferencing it without taking its ownership will only work if the type of the stored value implements the `Copy` trait. One possible solution to extend the set of supported types is to clone the old values instead of dereferencing them. While this would still limit us to using types that implement the `Clone` trait, this approach is less restrictive since we can still reason about heap-allocated data structures, which is not possible when relying on the `Copy` trait. Since Prusti does not support interior mutability at the time of writing this thesis, we can safely assume that a derived implementation of `Clone` or a reasonable manual implementation will result in the correct behavior at runtime. Fields of a struct containing values will be cloned or copied to the new struct. Any references it contains can safely point to the same data since Rust's ownership system ensures that any of their contents will remain unchanged.

Conditional Evaluation However, another issue arises with this encoding when old expressions are evaluated conditionally. This situation arises when a contract contains explicit branching, for instance, if-else or match statements, or due to short-circuiting of Boolean expressions.

To illustrate, we will consider a function – whose signature and specification are presented in Listing 3.8 – that removes an element from a vector if the provided index is within bounds.

If we attempt to naively evaluate and store the contents of the old expression before calling this function, we encounter a panic if the index is out of bounds. To resolve this issue, we adopt a translation strategy proposed in the Master's thesis by Mayer [4]. The fundamental premise of this translation is that old expressions can be rewritten by "pushing" them inwards until they are only applied to the parameters of the function the contract was attached to. For example when dealing with an expression of the form $old(E(p_1, .., p_n))$ where E is an arbitrary expression and p_i are the parameters of the parent function occurring within this expression, it is equivalent to $E(old(p_1), .., old(p_n))$.

While providing a formal proof for this equivalence applied to Rust expressions in Prusti is beyond the scope of this thesis, we offer an informal argument below. Prusti's contracts require that every function call within a specification is pure. This implies that they are devoid of side effects and are deterministic. Consequently, we can interpret E as a function. Given this interpretation, it is implicit that the result of evaluating E is solely dependent on the values of its input parameters, as no external state or side effects influence the evaluation. Therefore, the equivalence of evaluating each parameter of the expression in its old state, i.e., $E(old(p_1), .., old(p_n))$, and evaluating the whole expression in its old state, i.e., $old(E(p_1, .., p_n))$, is a logical consequence of the purity constraints imposed by Prusti's contracts.

Under this assumption the contract for our prior example is rewritten to:

```
#[ensures(index < old(v).len() ==> old(v)[index] == result)]
```

After rewriting specifications according to this rule, the task of storing old expressions in their old state becomes more straightforward. Instead of evaluating potentially critical operations, we simply clone all function parameters that are used within an old expression. The downside of this approach is that any function parameters used within an old expression must implement `Clone`. Cloning the full argument might result in the duplication of large regions of memory, adversely affecting the performance of the program.

A call to the function `remove` from our previous example will be translated as follows[2]:

```
let old_v = v.clone();
let old_index = index.clone();
remove(v, index);
assert!(
    index < old_v.len() ==> old_v[old_index] == result
);
```

Encoding Move Semantics Note that we technically would not have to clone the variable `index` in our previous example, since it is not a mutable reference and it implements `Copy`. Without cloning it, however, if that value

[2]The fact that implication is not valid Rust syntax and not allowed outside of specifications is ignored for simplicity

was moved into the function instead of being copied, we would no longer be able to access it after the call since ownership is transferred to the callee. However, this problem is not specific to values that are used within old expressions, it applies to any moved value used in a specification! Additionally, a non-reference value can be modified within the called function. If this variable occurs within a function's postcondition or pledge, the specification refers to the variable's value before the function was called. This is part of Prusti's logic because non-reference values do not have a post state. Therefore, if a user refers to such a parameter in a postcondition or pledge, Prusti automatically interprets this as an old expression for user convenience.

So even if we were able to access the argument we no longer own, it might not contain the correct value anymore. Listing 3.9 showcases this problem. The program, absent the added assertion, is successfully verified by Prusti. The runtime check we generate with our current translation fails compilation because `s` becomes inaccessible. Yet, more crucially, even if it would successfully compile, the check would fail due to the modification in the length of `s` occurring within the function.

Listing 3.9 Example of function consuming its input and modifying it, with an incorrect runtime check.

```
#[ensures(result.len() == s.len() + extension.len())]
fn extend(mut s: String, extension: &str) -> String {
    s.push_str(extension);
    let result = s
    assert!(result.len() == s.len() + extension.len())
    s
}
```

Consequently, moved values in a postcondition or pledge must also be stored in their old state. Even in instances where values are not used within an old expression, for correct runtime evaluations, they need to be treated as such.

A correct runtime check for a call of the previous function has to be translated as follows:

```
let old_s = s.clone();
let result = extend(s, t);
assert!(result.len() == old_s.len() + t.len())
```

Although the parameter s is not used within an old expression, we still evaluate it in its old state. Note that this latest adjustment to the translation is only applied to postconditions and pledges. Inline assertions operate within a different scope, and any function parameters used within them must be evaluated in their current state unless they are part of an old expression.

Listing 3.10 Translations of quantifiers to checkable expressions, assuming domain of type can be iterated over.

```
// forall(x: T expr(x))              // exists(x:T expr(x))
{                                    {
    let mut holds_forall = true;         let mut exists = false;
    for x in dom_T {                     for x in dom_T {
        if !T(expr(x)) {                     if T(expr(x)) {
            holds_forall = false;                exists = true;
            break;                               break;
        }                                    }
    }                                    }
    holds_forall                         exists
}                                    }
```

Quantifiers

As presented in Chapter 2, Prusti supports both universal and existential quantifiers. Bound variables are constructed as arguments of a closure and must be annotated with a type. In this discourse, quantifier triggers will be disregarded. To simplify the translation, we first note that a quantifier with multiple arguments can be rewritten as a series of nested quantifiers when ignoring triggers. For instance, consider the quantifier:

```
forall(|x:T, y:S| expr(x,y))
```

which can be rewritten as:

```
forall(|x:T| forall(|y:S| expr(x,y)))
```

Moving forward, we will restrict our considerations to quantifiers with a single argument. To rewrite a quantifier as an expression that can be evaluated at runtime, we must traverse the values of its domain. Assuming that all the values of a domain are contained in an `Iterator` named `dom_T`, we can rewrite quantifier expressions as shown in Listing 3.10. Leveraging Rust's functional characteristics, enclosing these blocks of code within brackets allows us to use them as subexpressions in any overarching Boolean expression. Consequently, this translation supports nested quantifiers as well. Subsequently, our focus shifts to devising a strategy that enables the traversal of all the values within a domain.

Types The bound variables of a quantifier are always annotated with a type to define their domain. For the iteration over every value within the domain of a primitive integer type `T`, we can simply instantiate the variable `dom_T` of our previous translation with the range `T::MIN..=T::MAX`. While other types are supported by Prusti, our translation does not support them. The reasons for this will be laid out in Chapter 4 and possible extensions will be discussed in Chapter 6.

Bounds The process of iterating over all values of a type's domain is highly inefficient. Especially for nested quantifiers or quantifiers over large types, performing runtime checks will oftentimes be infeasible in practice. However, it is very common for developers to specify bounds on the expression used within a quantifier. Although Prusti does not currently support explicit declaration of quantifier bounds, it is oftentimes possible to infer them from the contained expression.

In Prusti, bounds over universal quantifiers are typically defined by using implications, declaring that a condition needs to be met only if a certain bound condition is satisfied. For instance, to make assertions about the contents of a collection, specifications are generally formulated as:

```
forall(|i: usize| i < v.len() ==> check(v.lookup(i)))
```

Since implications are vacuously satisfied for all values not meeting the bound condition, checking values outside of the bounds at runtime is unnecessary. In the given example we can limit `dom_T` to the range `0..v.len()`. Similarly, the domain of an existential quantifier is usually limited using a conjunction of bound conditions and the condition to be checked itself. Our strategy is to derive a lower and an upper bound for each quantifier if possible, to then iterate over the range between them.

3.2 Verification-based optimizations

In this subsequent segment of the thesis, the objective is to exploit the additional information introduced through the annotation of contracts combined with the advanced verification capabilities of Prusti to enable optimizations that may be unattainable for a conventional compiler.

3.2.1 Dead Code Elimination

By adjusting Prusti's encoding of Rust programs, we can employ it to pinpoint unreachable code. More precisely, the statement `prusti_refute!(false)` will result in a verification error if it is located in an unreachable segment. By embedding a refutation of false at the beginning of every block of code that is the target of a branching operation, we can identify blocks that are unreachable, subsequently allowing their removal by simplifying or removing the branching structures. Listing 3.11 showcases the encoding of a match statement with one unreachable arm due to the function's precondition. Prusti will report an error for the refutation on line 13, which allows us to remove this branch.

Listing 3.11 A function with an unreachable match arm (because of its precondition), with added refutations to detect unreachable code.

```rust
#[requires(x % 2 == 0)]
fn foo(x: i32) -> i32 {
    match x {
        x if x % 4 == 0 => {
            prusti_refute!(false);
            3
        },
        x if x % 7 == 0 => { // this case can be removed!
            prusti_refute!(false);
            6
        },
        _ => {
            prusti_refute!(false);
            0
        }
    }
}
```

3.2.2 Removing Unneeded Checks

Rust performs a variety of checks at runtime to ensure memory safety and prevent undefined behavior. Examples include overflow checking for certain operations or boundary checks during the access of arrays or slices. Frequently, Prusti is able to prove that these operations are secure, deeming the associated check redundant. In such instances, we can remove these checks and modify the checked operations into their unchecked equivalent. The explanations of these optimizations are specific to the MIR, which is why most information regarding this topic will be presented in Chapter 4.

Chapter 4

Implementation

4.1 Runtime Checks

For the insertion of runtime checks into the produced executables, we make modifications on two stages of the compilation. In particular, we modify Rust's AST using macros and the MIR by overriding compiler queries. We rely on the modifications to both representations because each of them offers certain advantages.

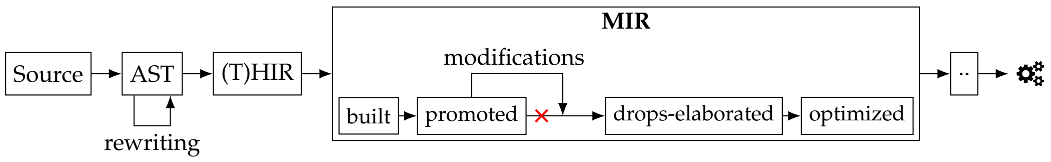
The AST is modified to generate functions that can be used to check specifications and translate expressions. The advantages of modifying the AST are that generating a lot of new code is significantly less tedious and our generated code will still be type and memory safe. The nested structure of expressions in the AST additionally facilitates translations that would be a lot more challenging during later stages.

On the MIR level we will, among a few other things, identify function declarations and calls with associated contracts and insert calls to the previously generated check functions. The advantages of making modifications at this stage include:

- All names are resolved and full type information is available.
- The control flow is explicit and the compiler offers functionality to analyze it.
- Expressions and Statements are simpler since they are no longer nested.
- The borrow checking information is available.

An overview of the performed modifications is shown in Figure 4.1.

Figure 4.1: Illustration of the modifications performed in the compiler pipeline.



4.1.1 AST Rewriting

Using Rust macros, we can modify the AST of a program during macro expansion. As discussed in Chapter 2, Prusti's contracts are used to generate specification functions using procedural macros. Prusti uses attribute macros like `#[ensures(..)]` which are attached to functions and function-like macros for inline assertions. We will first deal with the translation of contracts defined via attribute macros.

Again, we will use a function T to model the rewriting of expressions within contracts while first discussing the different kinds of contracts.

Function Generation

Analogous to the manner in which Prusti generates specification functions for each contract attached to a function (Section 2.2.1), we generate check functions for each specification we intend to check at runtime. Unlike specification functions which are solely utilized for verification purposes, check functions are designed to be executed and are intended to test their corresponding contracts. At this stage, these functions will only be declared but no calls to them will be inserted into the program.

The functions we generate to check contracts attached to a function are generally of the form:

```
#[spec_only]
#[check_only]
fn check_uuid(inputs) {
    let mut error_message =
        "Contract 'expr' was violated at runtime";
    if !(T(expr)) {
        panic!(error_message);
    }
}
```

The main challenges of this translation involve generating a function signature that gives access to all important data such as the result and values of parameters in their old state, and generating an expression that can be checked at runtime given the contents of the contract by implementing the rewriting function T.

The signature of a check function depends on the parameters of the function they are attached to and the kind of contracts they are checking. All check functions inherit all the parameters (including generic type parameters) from the function they are attached to. This already resolves the naming issue that we discussed in Chapter 3, since parameters have the same names as the variables occurring within the checked expression. For each pre- and postcondition, one check function is generated. For a pledge we generate two check functions, one for its left-hand side and another for its right-hand side. The error message that is passed to the panic macro is a string containing the source code of the expression that has failed at runtime.

All function names contain a UUID, allowing us to link them to the function they were attached to. Additionally, certain attributes are added to each check function, not all of which are shown in the above example. For instance, the attribute `#[spec_only]` is used to stop Prusti from attempting to verify such functions.

In the following, we look at the implementation of the translation of Prusti specifications.

Result

The `result` keyword can be used in postconditions and the left-hand side of pledges. To handle it, we extend the set of parameters for the check function of these contracts with a parameter of the same name. No modifications to the expression to be checked itself are necessary.

Old Expressions

To check contracts containing old expressions at runtime, we make modifications to both the check function's signature, as well as to the expression to be checked. The signature of the check function is extended with an `old_values` tuple with the same number of fields as the associated function has parameters. The i-th field of this tuple either has the same type as the i-th parameter of the function or is of type unit if the corresponding parameter never needs to be evaluated in its old state. The purpose of the unit types in this tuple is to reduce the memory overhead introduced by cloning arguments unnecessarily and to allow easy identification of the required parameters in the later stages of the translation.

To illustrate, we use the example shown in Listing 4.1. We first translate the contents of the given contract to an expression that can be evaluated at runtime. At first, at least conceptually, old expressions are moved "inwards"

Listing 4.1 An example function with a postcondition and multiple arguments.

```
#[derive(Clone)]
struct Simple(i32);

#[ensures(old(a.0 + b.0) == c.0)]
fn foo(a: &mut Simple, b: &Simple, c: Simple) {..}
```

according to the equivalence outlined in Chapter 3. For instance, in our running example, the contract would be rewritten to:

```
old(a).0 + old(b).0 == c.0
```

Afterward, we traverse the AST of the expression to be checked. For each encountered identifier, we determine whether it should be evaluated in its old state. We apply the previously discussed rules as follows:

- If the identifier is not one of the function's parameters, we leave it untouched in all cases.
- If the identifier is one of the function's parameters but is an immutable reference, we leave it untouched as well. Evaluating it in its old state is unnecessary because in the absence of interior mutability the contents of this reference cannot be modified. For instance, this rule applies to the example function's parameter `b`, where the old expression around `old(b)` can consequently be removed.
- If the identifier is used within an old expression, referring to a parameter that is not an immutable reference, we replace it with an access to the correct field of the `old_values` tuple. In particular, if an identifier `a_i` refers to the i-th argument of the function the contract is attached to, then the expression `old(a_i)` will be replaced with the tuple access `old_values.i`. In our example, this rule applies to the function parameter `a`, where the expression `old(a)` will be replaced with `old_values.0`.
- If the identifier is a function parameter and is not a reference, we have to assume that it is a moved value and therefore also evaluate it in its old state, even if there is no old expression surrounding it. The rewriting to a tuple access is done as explained before. In our example, this rule applies to the function parameter `c`, where the identifier will be replaced with `old_values.2`.

Note that to determine the type of a function parameter, we can only rely on the information that can be extracted from the tokens of the type annotations of a function signature. No type or name resolution has taken place in this phase of the compilation. We only recognize references as such, if their type annotation starts with an `&`. This results in two problems, the first of which is that we have to assume that all non-reference types are moved even though some of them might implement `Copy`, resulting in unnecessary cloning. The second problem occurs in the presence of type aliases, which can hide details of a type we rely on. For a type alias containing an immutable reference, this only leads to unnecessary cloning, since we interpret these types as potentially moved values. For mutable references, however, this can cause inconsistencies in our runtime checks, which is a bug that we have not been able to resolve yet. A possible solution would be to resolve old expressions completely in the MIR instead of the combined solution we currently employ.

Pattern matching within function signatures is not supported by our current approach. For example, if a function signature contains an argument of the form `SomeStruct{a, b}: SomeStruct`, we are not able to translate usages of the variable `a` to an access of the `old_values` tuple.

Applying all of the previously discussed rules to our current example results in the expression translated to:

```
old_values.0.0 + b.0 == old_values.2.0
```

Since only `a` and `c` are used in their old state the type of the `old_values` tuple in the check function signature is:

```
(&Simple, (), Simple)
```

The complete check function being generated in this example looks as follows:

```
fn check_uuid(a: &mut Simple, b: &Simple, mut c: Simple, result: (),
↪   old_values: (&Simple, (), Simple)) {
    if !(old_values.0.0 + b.0 == old_values.2.0) {
        panic!(error_message);
    }
}
```

As long as this function is called with the correct arguments, this will perform a correct check of the contract in question. The responsibility to clone arguments, invoke the check functions, and pass the correct arguments will all be resolved via modifications of the MIR.

Quantifiers

The implementation for the translation of quantifiers follows the translation we outlined back in Section 3.1.2, in particular the translation displayed in Listing 3.10. As mentioned, we limit the set of types to primitive integer types. In particular these include all unsigned integers (`u8`, .., `u128`, `usize`) and signed integers (`i8`, .., `i128`, isize).

Bounds Extraction To make runtime checks of quantifiers more efficient, we try to derive bounds outside of which checking the expression under analysis is unnecessary.

When dealing with a universal quantifier, we will first try to isolate the expression that potentially contains its bounds. If the body of the quantifier is of the form `!e1 || e2`, which is what an implication is turned into by the preparser (Section 2.2.1), then `e1` is the expression we use for trying to extract a bound. For existential quantifiers, the expression that potentially contains bounds is the quantifier's body itself.

In the following, we will assume that the variable in a quantifier is named `x`. To collect a set of bounds, we walk the AST of the expression recursively while following a set of rules as shown in Algorithm 1. The operation $\circ$ represents an integer comparison out of $\leq$, $<$, $\geq$ or $>$. A lower or upper bound is generated for each subexpression of the form $x \circ e1$. For conjunctions, we process each conjunct separately to try and derive multiple bounds. If we encounter a negated expression, the bound derived from that expression has to be inverted. However, if a negated expression contains further conjunctions, we cannot extract bounds from this expression. Otherwise an expression like `!(x < 5 && x > 10)` would result in us deriving a lower bound of 5 and an upper bound of 10, although the expression is valid for all values of `x`.

Algorithm 1 The recursive bound extraction process.

```
1: function EXTRACT($e$)
2:     if $e = x \circ e1$ then
3:         return [bound($e1$, $\circ$)]
4:     else if $e = a$ && $b$ then
5:         return EXTRACT($a$) $\oplus$ EXTRACT($b$)          ▷ List conjunction
6:     else if $e = !a \land \neg$IS_CONJUNCTION($a$) then
7:         return INVERT(EXTRACT($a$))                  ▷ Invert bound
8:     else
9:         return []                                    ▷ Empty list
10:    end if
11: end function
```

Note that this extraction and translation strategy does not work for quantifiers with multiple arguments. Consider the following example, specifying that `v` is a sorted collection:

```
forall(|i: usize, j:usize| i < j && j < v.len() ==> v[i] <= v[j])
```

With our implementation, we would try to create two nested loops iterating over possible values for `i` and `j`. For `i` we would determine `j` as its upper bound, however, if the outer loop iterates over `i`, then `j` will not even be defined yet. To solve this, our implementation would have to detect the transitive relation of `i` and `v.len()` since the correct range for `i` is `0..(v.len() - 1)`.

At the end of this process, we receive a set of bounds. For us to accept an extraction and generate runtime checks for them, we define some rules to ensure the encoding is correct, but also to protect the user from checking contracts that might run forever. We can have at most one upper and one lower bound for each quantifier. The range we loop through will be of the form `lower..upper`. If we found a lower bound for our quantifier, then `lower` will get the value of this bound, otherwise we use `Type::MIN`. The same thing is done for the upper bound, with a default value of `Type::MAX` instead. To make sure users do not generate runtime checks that run forever, we also require that any type larger than 8 bits must always have exactly one upper bound, and if these types are signed they also need a lower bound.

While there are better solutions and many interesting possible extensions of this implementation, they were unfortunately out of the scope of this thesis. Extracting the bounds of quantifiers is a similar problem to inferring invariants for loops with numeric iteration variables, which has been extensively studied in the literature. An important thing in this context is also, that a user should easily understand which cases are supported and which are not. Many solutions we considered would still only solve this problem for a specific subset of cases.

Manual Bound Annotations It is apparent that our bound extraction, as defined, is very limited. For quantifiers to be checked at runtime, they can only have one argument and their body must have the previously described structure for us to derive bounds. To give users more power when runtime checking quantifiers, we added a new custom attribute that quantifiers can be extended with, to manually declare the range for the runtime checks of a quantifier. An example of such an annotation – as it would be required to correctly check our previous example – is shown in the following:

```
forall(
    #[runtime_quantifier_bounds(0..=v.len()-1, i+1..=v.len())]
    |i: usize, j:usize| i < j && j < v.len() ==> v[i] <= v[j]
)
```

The arguments passed to this custom attribute will be used to generate the loop ranges of our translation. Their correctness is the responsibility of the user and will not be checked, but would be an interesting extension of this feature. By verifying that the negation of the imposed bound implies the body of the quantifier, we could be certain that no values outside of the bound can violate the condition.

Ownership and Freeing Memory

When considering how the check functions we are generating in this section will be invoked, there are some clear problems regarding ownership. For example, consider the following simple function with a precondition:

```
#[requires(v.len() == 5)]
fn foo(v: Vec<i32>) {..}
```

The check function that will be generated for this function will have the following signature:

```
fn check_pre_foo(v: Vec<i32>) {..}
```

If we encounter a call to this function and want to check its precondition, the check function would have to be invoked as follows:

```
let v = vec![1,2,3];
check_pre_foo(v);
foo(v);
```

The borrow checker would not accept this program, since `v` is moved into both functions. However, as we will see later, we are able to bypass this and still invoke both calls. When dealing with `Copy` types or references, this is safe since the check function only checks pure contracts. Additionally, non-reference arguments are only accessed through the `old_values` tuple in check functions. The problem with moved values located on the heap, however, is that the check function will free the moved region. The function itself, `foo` in our example, will do the same and try to free `v` a second time, leading to undefined behavior.

To avoid this, we stop check functions from freeing any of the arguments they inherited from their parent function, by invoking `core::mem::forget` on all of them.

However, this does not solve the problem completely. In cases where the specification expression moves the argument further, the call to `forget` results in a "use after move" error. We found that, nevertheless, this approach of forgetting arguments allows more contracts to be checked. A proper solution would require substantial changes in our approach, either disallowing specifications from capturing values completely – which would severely restrict the expressiveness of contracts – or abandoning the idea of generating check functions and checking contracts purely with modifications in the MIR instead.

Inline Assertions

Inline assertions are processed using function-like macros, as opposed to the attribute macros we considered in the previous section. For that reason,

Listing 4.2 Extension of Prusti's encoding of inline assertions for runtime checking.

```
prusti_assert!(expr);              if false {
                                       ... // original Prusti branch
                                   } else if true {
                                       #[check_only]
                                       #[spec_only]
                                       || -> bool { true };
                                       let mut error_message = "..";
                                       if !(T(expr)) {
                                           panic!(error_message);
                                       }
                                   }
```

instead of generating check functions for these kinds of specifications, we extend the original encoding of Prusti as shown in Listing 4.2.

The second branch of this encoding added to the original translation of Prusti if an inline assertion is checked at runtime. The closure that is defined at the beginning of this branch is used to mark this block with attributes that can be identified on the MIR level. Firstly, we can use it to mark this branch as specification only with the `#[spec_only]` attribute, so it does not influence Prusti's encoding and verification. Secondly, it is used to mark this block as a check block, allowing us to make further modifications on the MIR level later.

The important difference is that when a procedural function-like macro like `prusti_assert!(expr)` is expanded, we only operate on the tokens directly passed to it and have no access to the tokens of the function it is called from. Most importantly, we cannot determine which of the identifiers occurring within the passed expression are function parameters, which is a problem for evaluating old expressions. In this setting, we cannot rely on an `old_values` tuple, since we cannot derive its type. Instead, we shift the effort for resolving old expressions to the modifications we will be making on the MIR level and leave old expressions untouched in this phase. The translation of quantifiers still happens exactly as for function-level specifications.

Predicates

In the original state of Prusti, predicates could not be evaluated at runtime because their content is moved into a specification function and their body is replaced with an `unimplemented` statement. To enable the use of a predicate within specifications that are checked at runtime, we have to replace the body of the predicate with its translated content. Since old expressions cannot be used within predicates, only quantifiers will be rewritten here.

Error Reporting

The error messages as we described them so far allow a user to identify which contract has failed at runtime. In this section, we further adjust the translation of expressions to offer more precise error messages in the context of conjunctions. If a universal quantifier or a series of "and" operations yield a false result, there is a single sub-expression or condition within the sequence that is responsible for the overall falsehood[1].

To identify the failing conjuncts and improve the generated error message, we define the following function `check_expr`:

```
pub fn check_expr(expr: bool, added_info: &str, message: &mut String)
    -> bool
{
    if !expr {
        message.push_str(added_info);
        false
    } else {
        true
    }
}
```

This function will extend the error message we report to the user with the information contained in `added_info` if the expression that is passed to it yields false, and subsequently returns the Boolean value that was passed to it. A conjunction of the form `a && b` can be rewritten to:

```
check(a, "a was violated", &mut error_message)
&& check(b, "b was violated", &mut error_message)
```

Evaluating this expression has the same result as the original, but will extend the error message we report to the user with more precise information. Similarly, we extend the failing branch of a universal quantifier with a modification to the `error_message` string as well, reporting the value of the bound variable for which the quantifier was violated:

```
for x in .. {
    if !expr {
        error_message.push_str(
            format!("'expr' was violated for index x={}", x).as_str()
        );
        holds_forall = false;
        break;
    }
}
```

These modifications are only applied to the outermost chain of conjunctions of an expression. For instance, when translating the expression `!(a && b)`,

[1]While multiple sub-expressions might be false, the evaluation stops after encountering the first one due to short-circuiting and our translation of `forall`.

the contained conjunction will be left untouched since an error message stating that `a` or `b` was violated might be more confusing than helpful.

4.1.2 MIR Modifications

In this section we will discuss how modifications to the MIR are made in general, and subsequently how we can leverage this capability to properly check contracts at runtime.

As mentioned previously, the MIR of a program can be modified by overriding a specific compiler query. However, the Rust compiler interface did not quite support the modifications required for our purposes at the time we started this thesis. In the Rust compiler interface's original state, if we wanted to modify the result of the query `mir_drops_elaborated_and_const_checked`, we would first invoke the so-called *base query* to get the original result of the query and then modify it. For our purposes, however, we had a few requirements.

Firstly, we do not want to modify the result of the `mir_promoted` query, since its result is used for Prusti's encoding and modifications would influence its result.

Secondly, we rely on the results of the borrow checker, in particular the information about expiration locations. For this information to be computed, `mir_promoted` must have been constructed already. Once a MIR phase is constructed, we can no longer modify it since its result is cached. Looking at the compiler pipeline, this would lead us to the conclusion that `mir_drops_elaborated` is the query that should be modified. However, the representation obtained through the base query has already gone through various MIR passes and the information about expiration locations from the borrow checker is no longer accurate for this representation. Also, any information obtained through verification relating to specific MIR locations (which will be important in the last section of this chapter) might now be offset as well.

Instead of relying on base queries, we have to be able to fully re-implement the `mir_drops_elaborated` query. This would allow us to invoke the `mir_promoted` query and steal it, then perform our own modifications, and finally perform the checks and passes that the compiler performs. Initially, this was not possible, since the processing done by the compiler was not accessible through the compiler interface. We opened a PR[2] for the Rust compiler that exposes this functionality, which was eventually accepted.

With the capability to modify the MIR, we can now discuss how the programs generated by our translation need to be further modified for contracts

[2]PR: https://github.com/rust-lang/rust/pull/114628

to be correctly checked at runtime. Our overridden query is automatically invoked by the compiler for a specific item. Modifications are necessary if the item itself has pre or postconditions to be checked, or if its body contains calls to functions with any checked contracts or inline assertions. In the following, we will explain the modifications that are required for each of these cases.

Empty Start Block

When we create new blocks, they will always have the highest index within the current CFG. Since certain modifications require us to add basic blocks at the start of the function, before any of its actual code is executed, we move the original starting block at index 0 to a new block. Instead, we place an empty block at the start index with a `Goto` terminator jumping to the original start block. Additionally, we also adjust all terminators in the remaining body that point to block 0. This is more of a precaution since the starting block does usually not have back edges. This will later allow us to add blocks at the beginning of the function, i.e. right after our empty "dummy" block, by simply adjusting its terminator instead of having to move blocks around and constantly having to adjust other terminators pointing to blocks that were moved.

Even though modifications of this nature make the CFG more complex by adding empty, technically unnecessary blocks, they will eventually be optimized away again by later passes of the compiler.

Preconditions

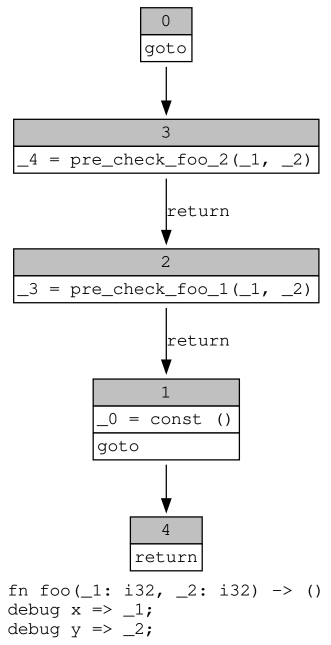
Given the body of a function, we first check if the function itself has preconditions that should be checked. If that is the case, we create a new block that contains a `Call` terminator to the check function of each precondition. The arguments of the current function are simply forwarded to the check functions and no additional arguments are required. The targets of each terminator need to be arranged such that they build a chain of calls where the final call jumps back to the actual function after terminating. Afterward, we set the target of the dummy block to the start of that chain. An example of an empty function with two preconditions and its modified MIR body is shown in Figure 4.2

Subsequently, we traverse the body of the function and look for `Call` terminators. Calls to local functions[3] can safely be ignored since their preconditions will be checked within their body. For all other calls, we also check if they have any preconditions to be checked. To prepend a precondition check to a call, we modify the MIR as shown in Figure 4.3. In this example,

[3]Functions which are part of the same crate

Figure 4.2: Insertion of calls to check functions into function body with two preconditions.

```
#[requires(x == 42)]
#[requires(y == 42)]
fn foo(x: i32, y: i32) {}
```


0
goto
3
_4 = pre_check_foo_2(_1, _2)
return
2
_3 = pre_check_foo_1(_1, _2)
return
1
_0 = const ()
goto
4
return
fn foo(_1: i32, _2: i32) -> ()
debug x => _1;
debug y => _2;

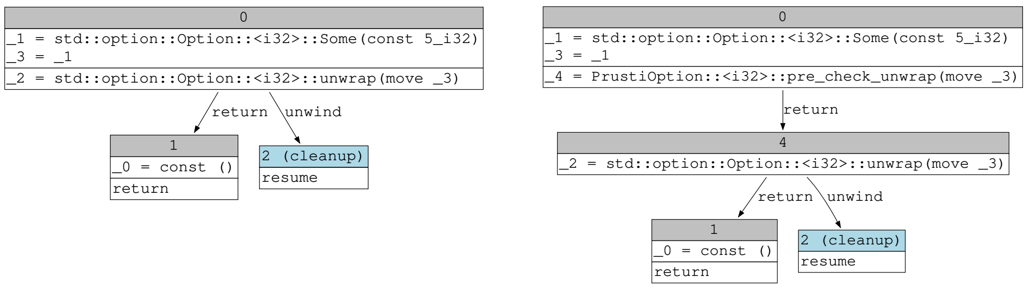
the call to `unwrap` has a precondition to be checked. Indeed, checking the precondition of `unwrap` is rather redundant, as the function itself conducts the same check internally. First, we create a new block (block 4 in the example) and move the original call into the new block. Afterward, we replace the terminator of the original caller block with a call to the check function and pass it the same arguments as the original call was invoked with. The target block of the check call is the newly created block.

With these modifications in place, each call to a function with a precondition will be checked correctly at runtime.

Postconditions

For postconditions, we currently only extend calls to functions with a postcondition with runtime checks, as opposed to inserting checks into the function body as well. Similarly to how we handle preconditions, we first traverse the CFG of an item and try to identify calls to functions with postconditions to be checked. The necessary modifications are slightly more involved in this case since the check function might require some of the arguments in their old state. Additionally, values that are cloned manually via

Figure 4.3: Prepending of check function for an external call with a precondition into MIR, before (left) and after (right).



```
_1 = std::option::Option::<i32>::Some(const 5_i32)
_3 = _1
_2 = std::option::Option::<i32>::unwrap(move _3)
```

```
_0 = const ()
return
```

```
resume
```

```
_1 = std::option::Option::<i32>::Some(const 5_i32)
_3 = _1
_4 = PrustiOption::<i32>::pre_check_unwrap(move _3)
```

```
_2 = std::option::Option::<i32>::unwrap(move _3)
```

```
_0 = const ()
return
```

```
resume
```

modifications of the MIR also need to be dropped again. At this stage, the compiler no longer automatically ensures that memory is properly cleaned up.

The modifications made can be summarized as follows:

- Before the function with a postcondition is called, clone its arguments if they are required in their old state.
- Call the actual function, as it occurred in the original MIR.
- Invoke the check function with the correct arguments, which now also include `result` and `old_values`.
- Drop the values that we cloned to free their memory.

To identify which arguments need to be stored before invoking the function, we analyze the signature of the check function, or more specifically the type of its `old_values` parameter. Recall that the `old_values` tuple contains fields for each function parameter, where the type of each field is either the unit type or the type of the corresponding parameter, depending on whether its value is required. Consider the example shown in Listing 4.3 of the wrapper function `remove` that removes an element from a `Vector`. The modifications made to the MIR for a call to this function are shown in Figure 4.4.

The type of `old_values` for the generated check function is:

```
old_values: (&Vec<i32>, ())
```

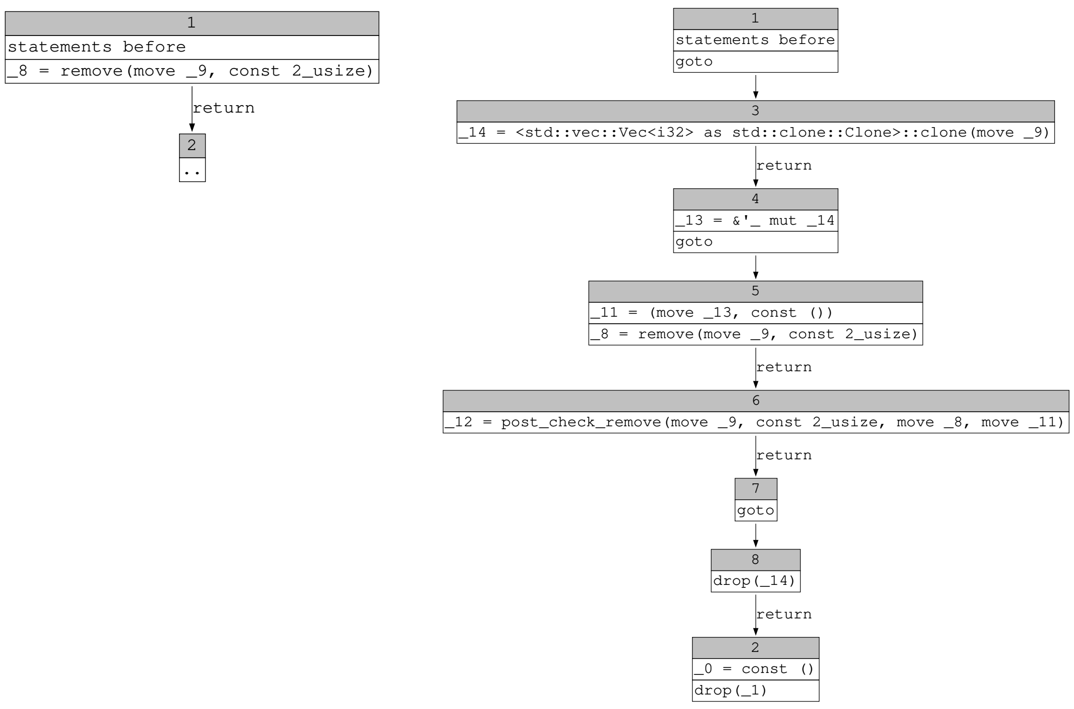
Therefore only the first argument passed to the function call needs to be cloned, which is the local _9 in our current example. As seen in the modified graph, we first clone the local _9 into a new temporary variable (block 3) and then store a reference to the result in a new local to match the type of the original value.

In block 5 we then construct the `old_values` tuple before calling the `remove` function, where its result is assigned to the local _8. Afterwards, the post-

Listing 4.3 Example function with postcondition.

```
#[trusted]
#[ensures(result == old(lookup(v, i)))]
fn remove(v: &mut Vec<i32>, i: usize) -> i32 {
    v.remove(i)
}
```

Figure 4.4: Insertion of a postcondition check into MIR for a function call, shown before (left) and after (right)



```
1
statements before
_8 = remove(move _9, const 2_usize)
    return
2
..
```

```
1
statements before
goto

3
_14 = <std::vec::Vec<i32> as std::clone::Clone>::clone(move _9)
    return
4
_13 = &'_ mut _14
goto

5
_11 = (move _13, const ())
_8 = remove(move _9, const 2_usize)
    return
6
_12 = post_check_remove(move _9, const 2_usize, move _8, move _11)
    return
7
goto

8
drop(_14)
    return
2
_0 = const ()
drop(_1)
```

condition is checked by invoking the check function (block 6). The provided arguments consist of the arguments that were also passed to `remove`, the result (_8), and the tuple we generated for the old values (_11).

Finally, we de-allocate the values we cloned earlier using the `Drop` terminator if necessary, as seen in block 8. Whether or not a value needs to be de-allocated depends on two factors. First of all, a value must only be dropped if it is located on the heap, which can be determined using the compiler interface. Secondly, if the value is not passed by reference but moved into the check function, then the check function will free it. Dropping the value again, after the check function has returned, would result in a double-free and lead to undefined behavior.

Pledges

For the insertion of runtime checks for pledges, we rely on the information of the borrow checker, in particular, the information derived through the Polonius algorithm. Prusti already uses this information and provides a suitable interface to determine the expiration locations of loans. We depend on a subset of its functionality, which we will shortly explain here:

- `get_call_loan_at_location`: If a location in the MIR contains a call to a function that returns a reference, this function will give us the identifier for the loan associated with the returned reference.
- `get_loans_dying_at`: Given a location in the MIR, this function will return all the loans that expire exactly after this location.
- `get_loans_dying_between`: Given two locations in the MIR with the second location being a successor of the first one, this function will return the list of loans expiring between them. Note that for a statement `s1` and its successor `s2`, invoking `info.get_loans_dying_between(s1, s2)` yields the same result as `get_loans_dying_at(s1)` However, for terminators with multiple targets and therefore multiple successor locations, a loan might only expire on a specific edge. In that case, `get_loans_dying_at` will not contain this loan, whereas `get_loans_dying_between` provides us with more precise information.

As previously mentioned, pledges can only be checked at the call site. In all examples, we will consider calls to the function `get_mut` shown in Listing 4.4 that we have already seen in previous examples.

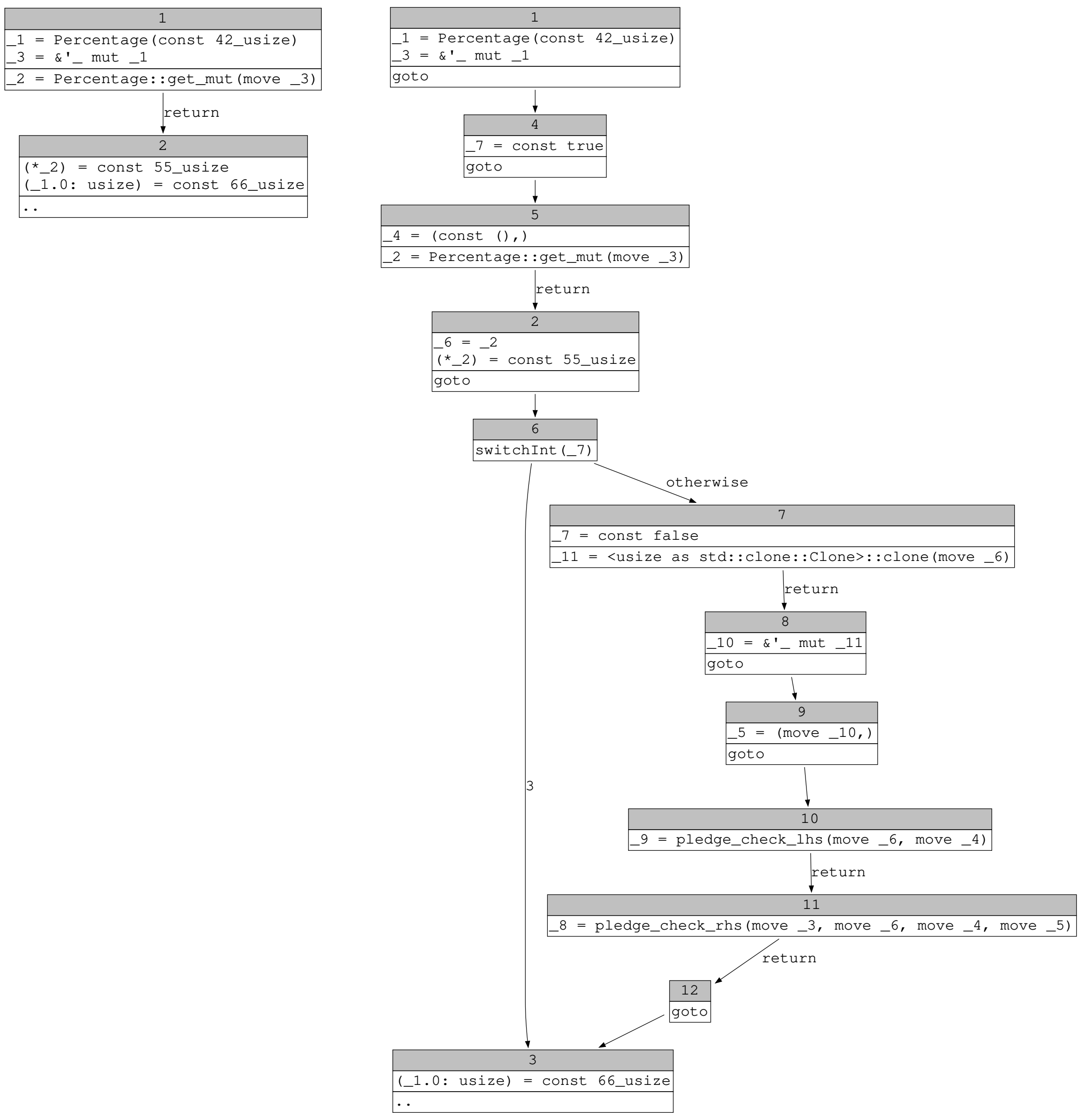
We will first consider the simple example shown in Listing 4.5. The corresponding MIR is shown in Figure 4.5. In Chapter 3 we described how the source code would have to be modified to check this pledge, now we describe how the equivalent modifications can be applied to the MIR.

Listing 4.4 Example of a pledge function.

```
struct Percentage(usize);

impl Percentage{
    #[assert_on_expiry(
        *result <= 100,
        before_expiry(*result) == self.0 && self.0 <= 100
    )]
    fn get_mut(&mut self) -> &mut usize {
        &mut self.0
    }
}
```

Listing 4.5 Simple call to pledge function defined in Listing 4.4

```
let mut p = Percentage(42);
let r = p.get_mut();
*r = 55;
// expiration location
p.0 = 66;
```



```
1
_1 = Percentage(const 42_usize)
_3 = &'_ mut _1
_2 = Percentage::get_mut(move _3)
  return
2
(*_2) = const 55_usize
(_1.0: usize) = const 66_usize
..
```

```
1
_1 = Percentage(const 42_usize)
_3 = &'_ mut _1
goto

4
_7 = const true
goto

5
_4 = (const (),)
_2 = Percentage::get_mut(move _3)
  return
2
_6 = _2
(*_2) = const 55_usize
goto

6
switchInt(_7)
  otherwise -> 7
  3 -> 3

7
_7 = const false
_11 = <usize as std::clone::Clone>::clone(move _6)
  return
8
_10 = &'_ mut _11
goto

9
_5 = (move _10,)
goto

10
_9 = pledge_check_lhs(move _6, move _4)
  return
11
_8 = pledge_check_rhs(move _3, move _6, move _4, move _5)
  return
12
goto

3
(_1.0: usize) = const 66_usize
..
```

Figure 4.5: Simplified MIR of Listing 4.5 before modifications (left) and after (right)

The function `get_mut` is invoked at the location `bb1[2]`, and we obtain the pledge associated with this call using `get_call_loan_at_location`. Afterward, we look for the expiration locations of this loan using `get_loans_dying_at` for each location of the MIR, in this case returning the location `bb2[0]`, the location where the last assignment to the returned reference occurs and right before `self` is accessed again. The check of the pledge needs to occur between these two statements, which is why this block is split into blocks 2 and 3 in the modified MIR.

Block 4 is added before the call to `get_mut`, assigning `true` to the local _7 to set the guard associated with the pledge. Additionally, in block 5 we construct the `old_values` tuple _2 which is empty in the current example. Otherwise, the cloning of old values would also occur before the call of the pledge function. After `get_mut` has been called, we copy its result into a new local _6 to handle the case where the result _2 is overwritten before the loan associated with it expires. This can occur due to reborrowing as we have seen earlier.

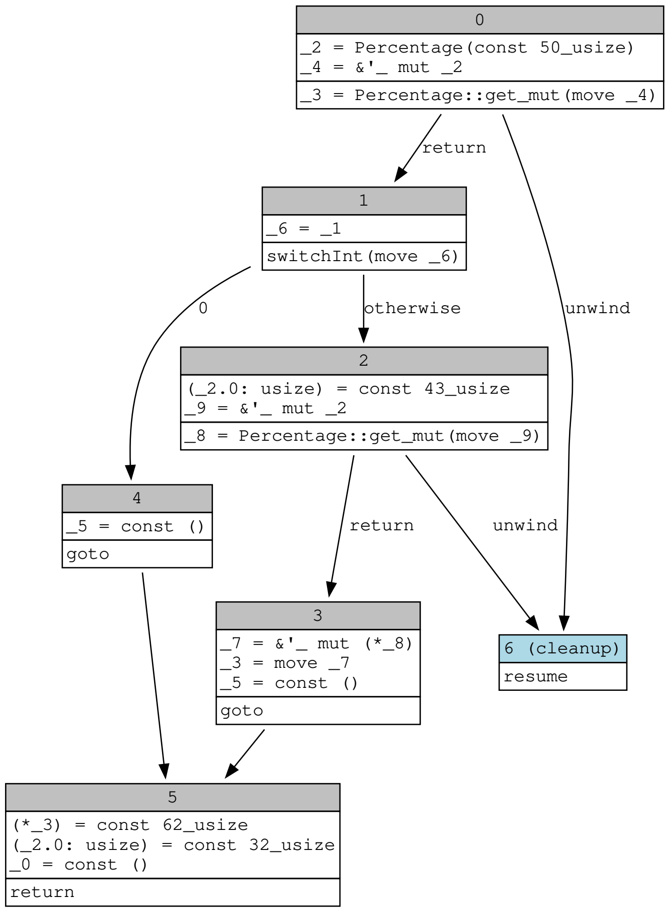
Between blocks 2 and 3 the actual checks are performed. The `switchInt` terminator in block 6 makes sure the checks are only performed if the guard _7 is set. In block 7 the guard is set to false, and we clone the result of `get_mut` for the value of `before_expiry(result)`. Just like old values are cloned before a function with a contract is invoked, we clone and borrow the result of the function for the `before_expiry` value before it expires in blocks 7 and 8, and create the `before_expiry` tuple in block 9. Afterward, the left-hand side of the pledge is invoked in block 10, and the right-hand side in block 11.

To show why the expiration locations obtained through `get_loans_dying_at` are not always sufficient, we consider another example using `get_mut` as shown in Listing 4.6. The unmodified MIR of this snippet is shown in Figure 4.6.

Listing 4.6 Conditional expiration of pledge defined in Listing 4.4.

```
fn foo(b: bool) {
    let mut p = Percentage(50);
    let mut r = p.get_mut(); // (L1)
    if b {
        // L1 expires here, if this path is taken
        p.0 = 43;
        r = p.get_mut(); // (L2)
    }
    *r = 62;
    // either L1 or L2 expires here
    p.0 = 32;
}
```

Figure 4.6: The unmodified MIR of Listing 4.6.


```
0
_2 = Percentage(const 50_usize)
_4 = &'_ mut _2
_3 = Percentage::get_mut(move _4)

1
_6 = _1
switchInt(move _6)

2
(_2.0: usize) = const 43_usize
_9 = &'_ mut _2
_8 = Percentage::get_mut(move _9)

4
_5 = const ()
goto

3
_7 = &'_ mut (*_8)
_3 = move _7
_5 = const ()
goto

6 (cleanup)
resume

5
(*_3) = const 62_usize
(_2.0: usize) = const 32_usize
_0 = const ()
return
```

In this example, the loan L_1 associated with the first call to `get_mut` *can* expire at one of the two locations `bb1[5]` or `bb5[2]`. The corresponding locations in the source file are marked with comments. However, the loan only expires at `bb1[5]`, if the `otherwise` branch of the `switchInt` terminator at this location is taken. Invoking `get_loans_dying_at(bb1[5])` will not return L_1, but by using `get_loans_dying_between(bb1[5], bb2[0])` we can obtain this information. To find all expiration locations we therefore not only check all locations of the MIR, but also each pair of `switchInt` terminators and their successors.

To insert runtime checks for such an example, we need to adjust the `switchInt` terminators targets and for the edges on which a loan expires, we jump to our check blocks instead of its original target. The full translation of this MIR is too involved to be discussed here, but the interested reader can refer to Appendix A.1.

Note that inserting multiple runtime checks at different locations can render other expiration locations invalid because of the modifications of the CFG. To avoid this, all locations where modifications need to be performed are first collected and sorted in decreasing order. Additionally, for the same reason, the insertion of runtime checks for pledges is always the first modification that is performed, because the insertion of checks for other contracts would lead to the expiration locations reported by the borrow checker being wrong.

This example also illustrates why the guard of a pledge needs to be set to false after it is checked. During execution, if the loan L_1 expires after the `switchInt` terminator, we will still come across the location `bb2[0]`. Without the guard, the same pledge would be checked for a second time, long after its expiration.

Inline Assertions

As opposed to the contracts we have seen so far, inline assertions are already inserted into the code as discussed in Section 4.1.1. However, if they contain old expressions they will not be evaluated correctly without additional modifications in the MIR. Old expressions in Prusti are interpreted as function calls by the Rust compiler. In the MIR we can identify these calls, but since expressions are not nested anymore, it is harder to determine which locals were actually passed into the old expression. The task to be solved consists of: finding calls to `old` within code blocks that were marked as `#[check_only]` and analyzing the arguments passed to `old` and the locals they depend on, to properly clone the correct parameters of the parent function and replace the correct locals with their clones in the correct places.

The challenge of resolving old for inline assertions is illustrated in Listing 4.7 and the corresponding MIR blocks of the `old` calls in Figure 4.7. In the function `foo`, the function parameter is used within the old expression and should be evaluated in its state at the entry of the function, therefore requiring some modifications. In the function `bar` on the other hand, the locally defined Boolean variable `b` is used within the old expression. Although `b` is also computed using `x`, since `b` is not a function parameter, `x` should not be evaluated in its old state. Finally, the function `baz` also uses a locally declared variable `b`, but since `x` is declared within the old expression, it has to be evaluated in its old state. Of course, this example is rather artificial, and writing specifications in such a way is nonsensical. However, the similarity of the MIR generated by these examples illustrates the challenge of differentiating the three cases based on their CFG. Our goal is to identify the locals

Listing 4.7 Three examples of old expressions in `prusti_assert` statements.

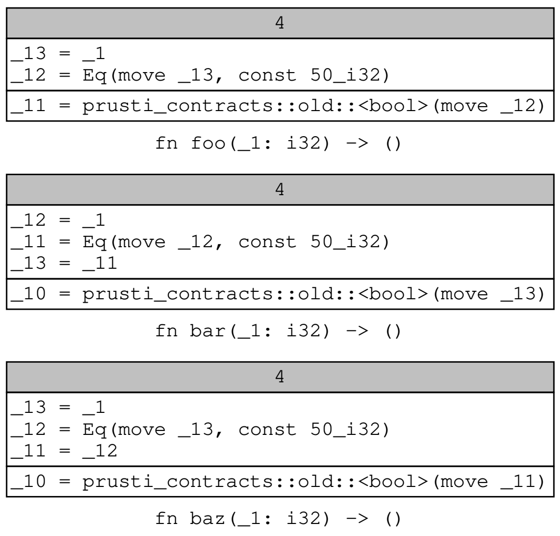
```
#[requires(x == 50)]
fn foo(mut x: i32) {
    x = 42;
    prusti_assert!(old(x == 50));
}

#[requires(x == 50)]
fn bar(mut x: i32) {
    x = 42;
    prusti_assert!({let b = x == 50; old(b)});
}

#[requires(x==50)]
fn baz(mut x: i32) {
    x = 42;
    prusti_assert!(old({let b = x == 50; b}))
}
```

of function parameters that were actually passed into the old expression, and eventually replace them with their clones.

To evaluate the correct parameters in their old state, we build a dependency graph for each local that is passed to an old function. A local depends on another, if there is an assignment to it where the other local occurs within its right-hand side or if the local is the destination of a call and the other local occurs within the arguments of that call. For each local defined in the MIR, we have access to debug information, used to determine whether a local belongs to a user-declared variable in the source code or if it is a temporary variable generated by the compiler. Additionally, we also rely on the source spans of the declaration of a local, telling us whether a variable was declared within the old expression or outside of it. The dependency graphs for each of the locals that are passed to old are shown in Figure 4.8.

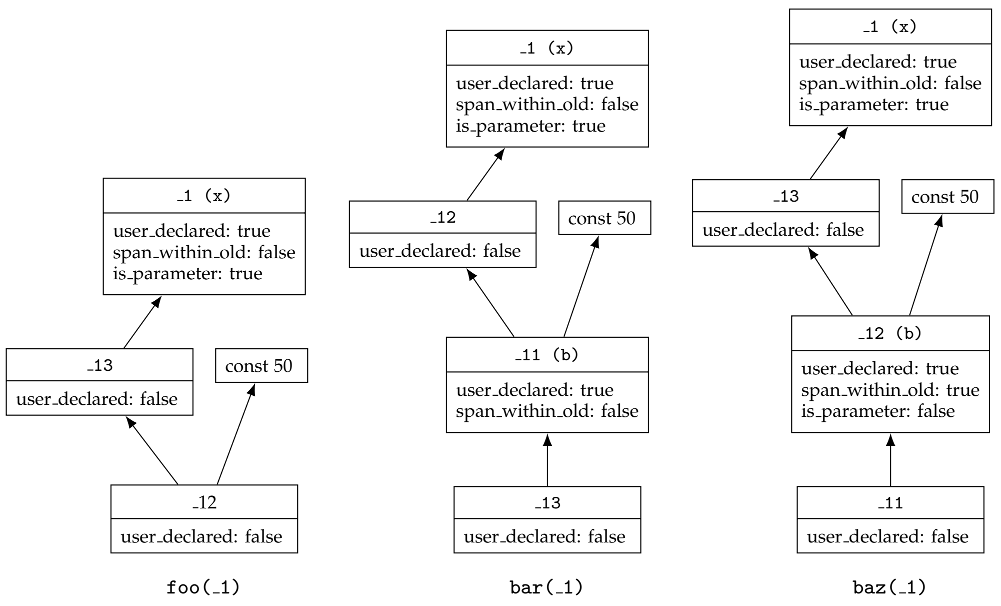
To determine the necessary modifications, the dependency graph is traversed starting at the local that was passed to the `old` function. If we encounter a constant or a variable that is user-declared and was not defined within the span of the `old` call, we can stop searching in this branch. For instance in the dependency graph for the function `bar`, we stop traversing the graph at the local _12 and the constant 50. Since we did not encounter any function parameters, this function does not require any modifications. For locals that are not user-declared or declared within the span of `old`, we visit the graph further. For example in the graph of `baz`, we encounter the user declared local _12 representing the Boolean `b`. Since `b` was declared within the `old` expression, we still visit the remaining dependencies. If we encounter a dependency on a function parameter, we are now certain that it

Figure 4.7: MIR blocks invoking the `old` function, for assertions as defined in Listing 4.7.



```
4
_13 = _1
_12 = Eq(move _13, const 50_i32)
_11 = prusti_contracts::old::<bool>(move _12)
fn foo(_1: i32) -> ()

4
_12 = _1
_11 = Eq(move _12, const 50_i32)
_13 = _11
_10 = prusti_contracts::old::<bool>(move _13)
fn bar(_1: i32) -> ()

4
_13 = _1
_12 = Eq(move _13, const 50_i32)
_11 = _12
_10 = prusti_contracts::old::<bool>(move _11)
fn baz(_1: i32) -> ()
```

Figure 4.8: Dependency Graphs for the locals passed to `old`, built from the MIR shown in Figure 4.7 and its debug information.



needs to be evaluated in its old state.

Each function parameter that is used in its old state will be cloned into a new local upon entry of the function. Then the uses of the original function parameter will be replaced with the cloned function parameter in the MIR locations where we detected the dependency. For example, assume that the result of cloning `_1` is stored in `_2` in both functions `foo` and `baz`. In both examples we would replace the first instruction `_13 = _1` in Figure 4.7, with `_13 = _2`. This modification ensures the correct resolution of any `old` expressions in inline assertions. Finally, if any of the cloned values are located on the heap, we have to make sure they are dropped again before the function returns. An example of the fully translated MIR for the function `foo` can be found in Appendix A.2.

4.2 Verification-based Optimizations

To perform optimizations based on the result of verification, we employ the previously explained modifications to the MIR by overriding `mir_drops_elaborated_and_const_checked`. Now these modifications are based on errors reported by Viper, which can be linked back to specific MIR locations.

4.2.1 Dead Code Elimination

In Chapter 3, the idea of inserting `prusti_refute!()` statements into the targets of branching operations was explained. Instead of inserting these refutations into the actual Rust programs, they are inserted into Prusti's Viper encoding. The encoded Viper programs generated by Prusti are organized in labeled blocks for each basic block in the MIR. If the optimizations of this section are enabled, we extend this encoding with `refute false` statements in each Viper block that is the result of encoding a MIR block that is the target of a `switchInt` terminator.

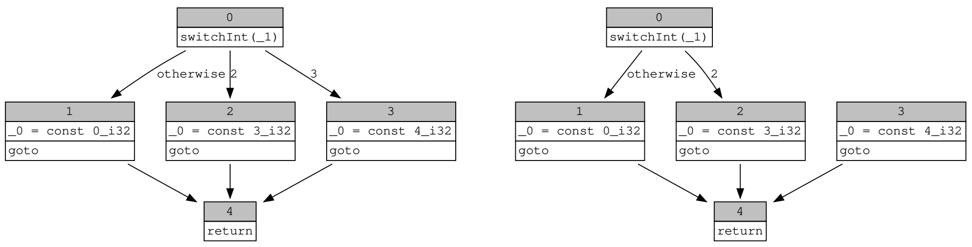
Additionally, we need to ensure that Viper is configured to report all errors it can detect and does not stop at the first error it encounters. After the verification of the encoded Viper program has concluded, the errors generated by the inserted refute statements are filtered out, resulting in a set of unreachable blocks for each verified item in the MIR. Other errors are still reported to the user.

Once the `mir_drops_elaborated` query is invoked for a specific item, we consult the list of unreachable locations to modify it accordingly. The only necessary adjustments in the MIR are modifications of the jumplists of `switchInt` terminators. For each `switchInt` terminator in the MIR, every target in the jumplist is eliminated if it was previously marked as unreach-

Listing 4.8 Example function with one unreachable match arm because of its precondition.

```
#[requires(x % 2 == 0)]
fn foo(x: i32) -> i32 {
    match x {
        2 => 3,
        3 => 4,
        _ => 0
    }
}
```

Figure 4.9: MIR of Listing 4.8 before and after optimization.



able during verification. If the `otherwise` target is unreachable, one of the targets in the jumplist is turned into the `otherwise` target. If only one target is left, the `switchInt` terminator can be turned into a `goto` terminator. If all targets are eliminated, implying that the `switchInt` terminator itself is unreachable, we transform it into a `unreachable` terminator.

Nodes are not explicitly deleted, but if all incoming edges of a node are removed, the later passes of the compiler will ensure that the node itself will be deleted as well.

A very simple example is shown in Listing 4.8, where the second match arm is unreachable because of the function's precondition. With the adjusted encoding, the result of verifying this function will identify block 3 in the MIR (Figure 4.9) as unreachable, and subsequently delete the edge to it. Later optimizations of the Rust compiler will delete block 3 completely.

4.2.2 Removing Assert Terminators and Unchecked Operations

As we have explained earlier, Rust generates `assert` terminators to check certain safety properties at runtime. Violations of these assertions result in programs panicking. Since Prusti is able to prove the absence of panics, these assertions can be eliminated in certain cases, further simplifying the control flow of functions and reducing the size of executables.

For example, for a division operation in Rust, the compiler will insert an assertion to prevent division by zero. An example of a basic block performing such a check is shown in the following:

```
0
_5 = Eq(_4, const 0_i32)
assert(!move _5, "attempt to divide `{}` by zero", _3)
```

If Prusti proves that an assertion can never fail, it can be transformed into a `goto` terminator.

These safety checks at runtime often involve the use of checked operations for arithmetic operations such as addition, subtraction, or multiplication. For example, an addition of two variables in the MIR is checked for overflows as shown here:

```
0
_5 = CheckedAdd(_3, _4)
assert(!move (_5.1: bool), "addition of .. would overflow")
```

The result of a checked operation is a tuple, where the first field holds the result of the operation and the second field contains a Boolean value that is true if the operation was successful. When Prusti is able to prove that an assertion associated with a checked operation can never fail, we not only remove this assertion but also transform the checked operation into an unchecked one. The block in the previous example is modified to:

```
0
_6 = Add(_3, _4)
goto
```

Additionally, all occurrences of `_5.0` in the remaining CFG need to be replaced with the new result of the unchecked operation, in this case, `_6`

Chapter 5

Evaluation

5.1 Runtime Checks

To evaluate the usefulness of our implementation, we first demonstrate how our implementation of runtime checks for Prusti contracts can be used in practice. We will then show a few examples and examine the errors that are generated for violated contracts. Afterward, we apply our implementation to the Prusti test set to determine how many typical Prusti annotated programs can be successfully checked at runtime.

5.1.1 User Guide

In this section, we will give an overview of how runtime checks can be used in Prusti. This part should also serve as a template for an extension of the Prusti user guide.

Setup

By default, Prusti stops after verifying a program and does not complete its compilation. For the purposes of runtime checking, we of course need to compile the program into an executable. When working with standalone Rust files using `prusti-rustc`, the Prusti flag `FULL_COMPILATION` must be set. When using `cargo-prusti`, the `CARGO_CMD` flag must be set to either "run" or "build".

To enable runtime checks, a user first has to set the Prusti flag `INSERT_RUNTIME_CHECKS` to one of the values "selective" or "all". If selective runtime checks are enabled, only contracts marked with the attribute `#[insert_runtime_check]` will be checked at runtime. With the option "all" enabled, all contracts are attempted to be checked at runtime. Enabling all runtime checks is not advisable in most practical use cases, since there are various Prusti features that are not properly supported yet.

Additionally, users have the option to enable debugging messages for runtime checks. By setting the flag `DEBUG_RUNTIME_CHECKS` to `true`, each check that is performed will emit a message stating which check has been executed. In the absence of this debug option, only breaches in contracts – which lead to panics – are visible to the user.

Supported Contracts

Function Contracts Runtime checks are supported for preconditions, postconditions, and pledges. The contracts can also be part of external specifications. For the `#[insert_runtime_check]` attribute to be applied to a method's contract, it must be placed in front of the contract. For example, for a precondition of a method to be checked at each invocation, it has to be annotated as follows:

```
#[insert_runtime_check]
#[requires(x % 2 == 0)]
fn foo(x: i32) {}
```

Inline Assertions Additionally, inline assertions such as `prusti_assume`, `prusti_assert` and `body_invariant` can be checked at runtime by adding the attribute to their body as follows:

```
prusti_assume!(#[insert_runtime_check] x % 2 == 0)
```

These statements can also be used within trusted methods, where they – despite being ignored by verification – allow users to write assertions using the Prusti specification syntax.

Predicates To be able to use predicates within contracts that are checked at runtime, they must be marked with the `#[insert_runtime_check]` attribute as well:

```
#[insert_runtime_check]
predicate! {
    fn is_even(x: i32) { x % 2 == 0 }
}

#[insert_runtime_check]
#[requires(is_even(x))]
fn foo(x: i32) {}
```

For obvious reasons, abstract predicates cannot be checked at runtime and will cause an error at compile time.

Specification Language

For expressions within contracts to be checked at runtime, only a subset of Prusti's specification language can be used, often with additional requirements.

Result The `result` keyword within postconditions and pledges can be used without any restrictions.

Old Referring to values in their old state is only possible if all function parameters that are used within the `old` expression implement the `Clone` trait. For example, consider the following method signature and its contract:

```
#[ensures(old(x.some_property()))]
fn bar(x: &mut SomeStruct) {}
```

Since `x` is used within the `old` expression and is a parameter of the `bar` function, the struct `SomeStruct` must implement `Clone` for this contract to be checkable at runtime.

Similarly, the before_expiry expression can only be used within checked pledges if the return type of the annotated function implements `Clone`.

Quantifiers For quantifiers to be checked at runtime there are various restrictions. The arguments of quantifiers can only have primitive integer types, i.e. `u8`, .., `u128`, `usize` and `i8`, .., `i128`, `isize`. To avoid runtime checks that run forever, quantifiers over all types larger than 8 bits must be bounded. Bounds can be introduced in two ways:

- For quantifiers with only one argument, bounds are automatically extracted from the body of the quantifier if possible. For example, consider the following quantifiers:

  ```
  forall(|x: i32| (x >= lb && x <= ub) ==> condition)
  exists(|x: i32| x >= lb && x <= ub && condition)
  ```

  The lower bound `lb` and upper bound `ub` will be determined automatically, and the conditions will only be checked within this range. The extraction only works if these expressions follow a certain structure. For universal quantifiers, the body must be of the form `bounds ==> condition` or `!bound || condition`. For existential quantifiers, the expression itself is used as the `bounds` expression. The `bounds` expression must be a conjunction containing one upper and one lower bound. For unsigned types, the lower bound is optional. A bound within a conjunction is only recognized as such if it is a comparison operation (<, <=, >, >=) where either the left-hand side or the right-hand side of the operation is the quantifier parameter, `x` in our example.

- Alternatively, quantifiers can be manually annotated with the ranges that should be used to check them at runtime. This feature's main purpose is to support quantifiers with multiple arguments, but can also be used in single-argument quantifiers. For example, a quantifier with two arguments can be annotated as follows:

```
forall(
    #[runtime_quantifier_bounds(0..10, x..=x+10)]
    |x: i32, y: i32| ..
)
```

The correct annotation of contracts is the responsibility of the user. An incorrect range might lead to passing checks for violated quantifiers, or unsubstantiated runtime errors in existential quantifiers. One advantage, however, is the fact that a user can willingly only check a smaller range of a quantifier to reduce the overhead of the performed check.

The presence of bounds still does not ensure that runtime checks for quantifiers are efficient and they should be used with caution.

5.1.2 Examples

In the following, we will present the output of a few examples to demonstrate the functionality of our implementation.

The first example, shown in Listing 5.1, contains a function `transfer` that has an incorrect postcondition. The function is called from the `main` method because otherwise, the insertion of runtime checks would have no effect. Running this program, after compiling it with Prusti, results in the following error:

```
thread 'main' panicked at account.rs:18:5:
Prusti Runtime Checks:
    Contract #[ensures(self.balance() == old(self.balance()) + amount)]
    was violated at runtime
```

The error is reported correctly and the failing contract can easily be identified. Of course, if this method was not trusted, verification would also allow us to identify the error easily.

Next, we consider the example shown in Listing 5.2 to demonstrate runtime checks for pledges, once again using the `Percentage` struct. Within the `main` method, we assign a value of `101` to the reference returned by the `get_mut` function, which violates the pledge. Compiling and running this program leads to the following error being generated:

```
thread 'main' panicked at pledge.rs:11:5:
Prusti Runtime Checks:
    Contract #[assert_on_expiry(*result <= 100, ..)] was violated
    at runtime
```

Listing 5.1 Example implementation of a `transfer` function with incorrect specification.

```
#[derive(Clone)]
struct Account {
    bal: u32,
}

impl Account {
    #[pure]
    fn balance(&self) -> u32 {
        self.bal
    }

    #[trusted]
    #[requires(amount <= self.balance())]
    #[ensures(self.balance() == old(self.balance()) + amount)]
    fn transfer(&mut self, other: &mut Account, amount: u32) {
        self.bal = self.bal - amount;
        other.bal = other.bal + amount;
    }
}

fn main() {
    let mut a1 = Account { bal: 10 };
    let mut a2 = Account { bal: 20 };
    a2.transfer(&mut a1, 12);
}
```

Listing 5.2 Example showcasing the violation of a pledge.

```
struct Percentage(usize);

impl Percentage {
    #[assert_on_expiry(
        *result <= 100,
        before_expiry(*result) == self.0 && self.0 <= 100
    )]
    fn get_mut(&mut self) -> &mut usize {
        &mut self.0
    }
}

#[trusted]
fn main() {
    let mut p = Percentage(42);
    let r = p.get_mut();
    *r = 101; // illegal assignment
    p.0 = 42;
}
```

Listing 5.3 Example of a prusti_assume statement, containing a predicate and a quantifier. The implementation of VecWrapper is omitted.

```
predicate! {
    fn smaller_than(x: i32, y: i32) -> bool {
        x < y
    }
}

#[trusted]
fn main() {
    let v = VecWrapper::new(vec![1,2,3,4,4,6,7]);
    prusti_assume!(forall(|i: usize| (i < v.len() - 1) ==>
        smaller_than(v.lookup(i), v.lookup(i+1))
    ));
}
```

From this error message, we can derive that the pledge attached to get_mut was violated at runtime. More precisely, we can tell the left-hand side of the pledge has been breached since only the left expression of the contract is displayed.

Finally, we consider the example shown in Listing 5.3, containing an inline assertion, a quantifier, and a predicate. The specified quantifier expresses that the provided VecWrapper is sorted, using the smaller_than predicate. However, the defined Vector is not strictly sorted, leading to the following error:

```
Prusti Runtime Checks:
Contract prusti_assume!(forall(|i: usize| (i < v.len() - 1) ==>
        smaller_than(v.lookup(i), v.lookup(i+1))
    )) was violated at runtime

> expression (i < v.len() - 1) ==>
        smaller_than(v.lookup(i), v.lookup(i+1))
        was violated for index i=3
```

The message not only contains the failed contract but also more precise information about the index that caused the quantifier to fail. The improvements to the reported errors, discussed in Section 4.1.1, simplify identifying the source of an error, improving the user experience.

5.1.3 Limitations

While we have touched on the limitations of our implementation in previous chapters, this section offers a concise recap and provides further details.

Specification Items Our implementation supports preconditions, postconditions, pledges, predicates, and inline assertions. Other kinds of specifications in Prusti, such as `prusti_refute!(..)` are not supported, but mostly because the properties that they express cannot be checked at runtime.

Moreover, functions with specifications to be checked at runtime cannot use pattern matching in their signature, which will result in an error after macro expansion. All values that are used in their old state, both explicitly with `old` expressions or when referring to moved values in postconditions or pledges, must be of a type that implements `Clone`.

Naming Issues A large portion of the performed translations are made using a syntactic representation of Rust programs, where the names of types, variables, and functions are not resolved yet. Consequently, certain keywords and patterns can lead to errors and must be avoided:

- Defining variables within specifications whose names conflict with a parameter of the function they are attached to can lead to type-checking errors or incorrect runtime checks.
- Defining functions with the same name as Prusti internal expressions, for example, `old` or `forall`, will cause them to be misinterpreted.
- Declaring type aliases that hide the mutability of a reference and using it in a function signature, can lead to incorrect runtime checks.

Ownership When attaching specifications to functions that take ownership of a value, this specification cannot move the value further. For example, a user cannot assign the moved parameter to a new variable via pattern matching.

Contract Contents Various features of the Prusti specification language cannot be properly evaluated at runtime and can therefore not be part of a specification that is checked. These features include:

- Snapshot equality and inequality
- The function `snap()`
- The function `model()`

Quantifiers The limitations of quantifiers were discussed in detail both in Section 4.1.1 and in Section 5.1.1. Without revisiting all the details, it is worth noting that only a subset of quantifiers can be checked at runtime, and these generated checks can substantially impact a program's performance.

5.1.4 Prusti Testset

To test how many typical Prusti annotated programs can be checked at runtime with our implementation, we apply the modifications to various programs in Prusti's test suite. We only examine the results of the verification and compilation.

We used Prusti's existing tests where the result of verification is examined[1]. UI tests were excluded since they examine the output of the AST rewriting, which causes all tests to fail expectedly since we rewrite the AST. In total, this test set included 824 test cases. For each test, we wanted to determine whether:

- the test case contains unsupported features that are not correctly encoded,
- the modified translation of the AST produces code that breaks the initial type checking,
- the modified translation of the AST changes the result of verification,
- the modifications to the MIR ever cause internal compiler errors

An overview of the results of these tests is shown in Figure 5.1. In 763 of the test cases, the verification had the expected result and compilation was successfully completed. A total of 61 tests resulted in errors. Note that the various kinds of errors listed below do not add up to the total number of errors, since some of the tests are responsible for more than one violation.

A total of 53 errors were caused during the rewriting of the AST. 26 of those tests contain unsupported features, in most cases because snapshot-equality or calls to `snap` were used. Note that the compilation continues in case unsupported features are encountered since they only cause a warning. Another 26 tests contain quantifiers that either had multiple arguments, an unsupported type, or no bound could be extracted for them. In one test case, we encountered the problem of pattern matching within the signature of an annotated function.

During type checking, a total of 12 tests were rejected due to a "use after move" error. The failures are caused by the problem of moved arguments within checks due to calls to `mem::forget`, as discussed in Section 4.1.1. Notably, a significant portion of these errors arise from function calls like `snap` or `snapshot_equality`, which, in any case, are not supported. Other function calls cannot cause these errors, because a function that captures a value is never pure and cannot be used within a contract. However, "use after move" errors can still manifest in supported contracts, but only when a moved value is reassigned to another variable within the specification. Out

[1]The directories verify, verify_partial, and verify_overflow in prusti-tests/tests/

Figure 5.1: Results of evaluating runtime check implementation on the Prusti test set.

Total	824
Successful	763
Total errors	61
Unsupported features	26
Failing quantifiers	26
Pattern matching in argument	1
Use of moved value	12
Non-clone type in old state	1
Encoding	7

of all the test cases, this scenario was observed in just one, supporting our argument that the solution using `mem::forget` is a decent compromise.

Unfortunately, our implementation still contains bugs related to the encoding of inline assertions. In a total of 7 cases, there were errors reported during the encoding of the program, most of them when programs were encoded using the "unsafe core proof". Unlike the previous cases, there is no technical justification for these errors and they will be fixed in the future.

Runtime Tests

To ensure the correctness of our implementation and to facilitate its future maintenance, we created a separate set of tests composed of 63 test cases, all of which are successful. For each of these tests, the program undergoes verification and compilation but is also executed. At runtime, we ensure that the checks for all contracts are performed and that only the violated conditions lead to a panic.

Prusti's test set was unsuitable for this type of testing because most of them only have empty `main` methods. To evaluate the behavior of runtime checks, the methods annotated with contracts must be executed. Extending all tests with applications of the functionality they define was not in the scope of this thesis.

5.2 Verification-based Optimizations

We now shift our focus to the second topic of this thesis: verification-based optimizations. In this domain, two specific types of optimizations were explored and implemented: dead code elimination and the removal of unnecessary assert terminators and checked operations.

Listing 5.4 Example function with unreachable blocks because of precondition.

```
#[requires(x % 15 == 1)]
pub fn foo(x: i128) -> i128 {
    let y = x * 2;
    if x % 3 == 0 {
        y + 1
    } else if x % 5 == 1 {
        // if precondition holds we always reach this branch
        x + 1
    } else {
        y - 1
    }
}
```

To activate these optimizations, the Prusti flag `REMOVE_DEAD_CODE` has to be set. As before, the modifications can only be examined if programs are compiled to completion.

5.2.1 Dead Code Elimination

The additional information introduced through specifications, and the logical reasoning capability of a static verifier allow us to identify unreachable paths in the MIR and eliminate them. As discussed previously, these modifications are realized by deleting outgoing edges of `switchInt` terminators in the MIR. Now we are going the inspect the effects of these modifications on the subsequent optimized stages of the MIR and the generated executables.

Consider the example shown in Listing 5.4. It showcases an example where the applicable optimization can lead to an extreme simplification of the function. When taking the precondition for granted, only the second branch of this function is reachable. Under this assumption, the function can be simplified to:

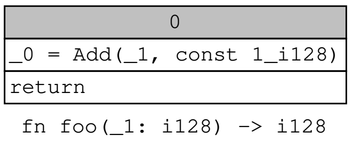
```
fn foo(x: i128) -> i128 { x }
```

The effect of the optimizations applied to this example becomes evident when examining the final optimized version of the MIR, now consisting of only a single operation, as depicted in Figure 5.2. In contrast, the version where only the default Rust optimizations were applied[2] retains 6 basic blocks. Proactively altering the MIR in a preliminary stage allows the Rust compiler to further remove unneeded operations.

Unsurprisingly, the effect of this optimization is also reflected in the performance of the program. On average, a call to the optimized function is

[2]All examples were compiled with optimization level 3

Figure 5.2: MIR of function `foo` defined in Listing 5.4 with dead code elimination enabled.



```
0
_0 = Add(_1, const 1_i128)
return
fn foo(_1: i128) -> i128
```

executed in just 2.1 nanoseconds[3], compared to the unoptimized version which takes 19.2 nanoseconds. However, it is important to approach these figures with a degree of caution, given the artificial nature of the provided example. In more realistic scenarios, only a minor fraction of the code might be identified as unreachable. In particular, this will often apply to code that is responsible for error handling, which can potentially be proven to be unnecessary through verification.

Moreover, this optimization is only correct as long as the specification of the function is satisfied. In case the previous function is ever called on a value that does not obey its precondition, the result of the optimized function will be inconsistent with its declaration.

5.2.2 Removing Assert Terminators and Unchecked Operations

While the process of dead code elimination can have a pronounced impact on the performance of Rust programs, the removal of assert terminators and unchecked operations results in a subtler effect. In release mode, Rust optimizes out many of the checked operations and their associated assert terminators to bolster performance. For operations such as addition or multiplication, overflow checks are only performed when compiling programs in debug mode. Therefore, the elimination of overflow checks is not a useful tool when optimizing for performance.

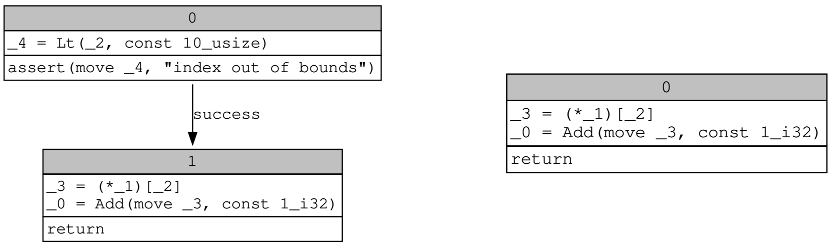
For other safety checks such as division by zero or slice indexing, however, Rust also tries to prove the safety of the operation before removing the associated check. Nevertheless, the additional information of contracts allows Prusti to remove assertions that the Rust compiler cannot. Consider the example shown in Listing 5.5, where an array is indexed using a function argument. Again, thanks to the precondition of the method `index`, Prusti determines that the bounds check for the array access cannot panic, allow-

[3]The measurements were averaged over a total of 10^9 invocations, on an Intel(R) Core(TM) i5-4460 CPU, 3.20GHz, Haswell

Listing 5.5 Access to an array that is safe because of its precondition.

```
#[requires(i < 10)]
fn index(a: &mut [i32; 10], i: usize) -> i32 {
    a[i] + 1
}
```

Figure 5.3: The MIR of the function `index` in Listing 5.5, with verification-based optimizations disabled (left) and enabled (right).



```
0
_4 = Lt(_2, const 10_usize)
assert(move _4, "index out of bounds")
```

```
1
_3 = (*_1)[_2]
_0 = Add(move _3, const 1_i32)
return
```

```
0
_3 = (*_1)[_2]
_0 = Add(move _3, const 1_i32)
return
```

ing us to eliminate it. The Rust compiler cannot perform this optimization. Figure 5.3 shows the final MIR stage for both cases. Because the function was compiled with optimization level 3, neither of the two CFGs contains any overflow checks, although the original function contained an addition. As expected, the bounds check was removed in the optimized MIR but still exists in the unoptimized MIR.

The examination of the generated LLVM[3] IR highlighted the effect of the optimization, reducing the number of statements from 6 down to 4. Interestingly, no substantial performance difference was observed between the generated executables. Despite the presence of a branching operation in the unoptimized version, we hypothesize that the consistent performance can be attributed to successful branch prediction. Given that we repeatedly invoke the function for benchmarking and every call follows an identical execution path, it provides an environment conducive to effective branch prediction. This is an inherent limitation on the effects of this optimization since every branch that can be eliminated will always exhibit this behavior in the unoptimized version of the program.

In more realistic use cases there might be a noticeable improvement in performance. Regrettably, a proper assessment of this optimization's performance impacts was not performed in this thesis. However, beyond mere performance enhancement, our modifications can also be viewed as a proof of concept for more nuanced safety checks at runtime. While the Rust compiler simply removes certain safety checks like overflow checks in release mode, our method ensures the removal of checks only when their safety is provably guaranteed. This strategy could provide a balanced tradeoff between performance and safety.

Chapter 6

Conclusion

This thesis' primary contribution is the design and realization of runtime checks for the contracts of the static Rust verifier Prusti. We detailed the high-level transformation of Prusti contracts into executable Rust constructs, enabling runtime evaluation of their logic. We implemented this translation and insertion of runtime checks through AST rewriting via procedural macros, and by directly interacting with the Rust compiler by modifying its MIR. Rust's safety properties both aided and challenged our endeavors. They allow for a simple interpretation of past states compared to other languages but also restrict the rewriting of specifications.

The work on this topic was qualitatively evaluated, using examples to highlight how runtime checks complement verification and help pinpoint errors in specifications. To further assess the robustness and applicability of our implementation, the compilation process of our implementation was tested against a large portion of Prusti's test suite. Furthermore, we incorporated a dedicated set of tests designed to evaluate the behavior of runtime checks in execution. We found that, despite the various limitations of our implementation, it can still be applied to a large portion of contracts.

Additionally, we explored the idea of verification-based optimizations through modifications of the MIR. In particular, we designed and implemented a technique for the detection and removal of dead code, and the elimination of superfluous safety checks. Although we could not perform a full-fledged performance evaluation of these optimizations, this work underscores the potential of static verifiers for not only ensuring code correctness, but also enhancing its efficiency.

6.1 Future Work

6.1.1 Contract Documentation in Runtime Check Failures

One feature that is currently under development in Prusti is the extension of contracts with supplementary documentation. This provides insight into the intent behind specific specifications. For example, a precondition could be extended as follows:

```
#[requires(x < 42, "x must be less than 42")]
```

This additional information could be incorporated and even extended when performing runtime checks. Ideally, the diagnostic messages could dynamically integrate the concrete values causing a violation by employing format strings in the documentation.

The importance of such enhancements becomes even more apparent in scenarios where a developer is working with external libraries. When a library has not been verified by the user, the meanings of specifications can be difficult to understand. By offering detailed diagnostic feedback phrased in plain language, rather than just logical expressions, developers can grasp the meaning behind contract violations more intuitively.

6.1.2 Improving Runtime Checks for Quantifiers

Several areas would benefit from improvements to quantifier checking. Specifically, the following topics warrant further refinement:

Extended Type Support Our current design only supports a heavily restricted set of types. This constraint arises because our translation is purely syntactic and has no ability to resolve types, and therefore no ability to reason about the contents of user-defined types. To extend the type support, we identify two possible strategies. Either, the translation would have to be performed on a lower level, for example, the MIR as well. However, this translation would be intricate and labor-intensive. Another possible strategy worth exploring is the idea of initiating an additional early run of the compiler, purely dedicated to extracting spanned type information. This information could then be used during the AST rewriting, facilitating more advanced translations. The issues outlined in Section 5.1.3 related to name resolutions could also be resolved with this approach.

Advanced Bound Derivation At present, the automatic derivation of boundaries is confined to quantifiers with a single argument containing expressions of very specific forms. Again, the fact that we are dealing with a syntactical representation complicates more sophisticated methods. However, if we were working with properly modeled logical expressions, we could employ more advanced tools to derive boundaries and determine transitive relations between arguments. Consequently, we could extract boundaries from more general kinds of expressions and determine the order of loops for quantifiers with multiple arguments more easily.

6.1.3 Replacing Functions with Equivalent Counterparts

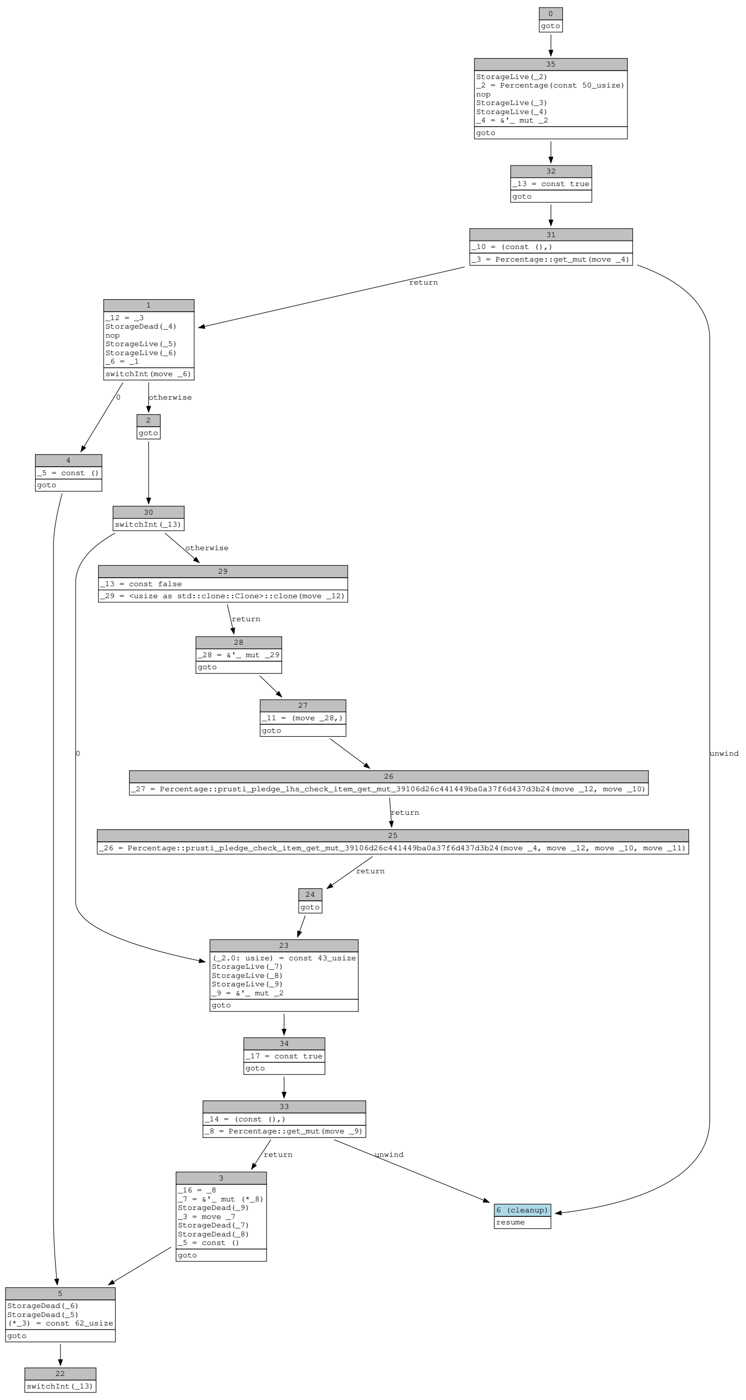
In Rust, there are many examples of different functions that exhibit the same behavior if a certain precondition is met but differ in performance. For instance, the functions of the `Option` enum `unwrap_or(default)` and `unwrap()` both return the same result if the input is of the variant `Some`. An interesting possible optimization through verification is the introduction of contracts that specify that a function call can be replaced with another if certain conditions are proven.

Appendix A

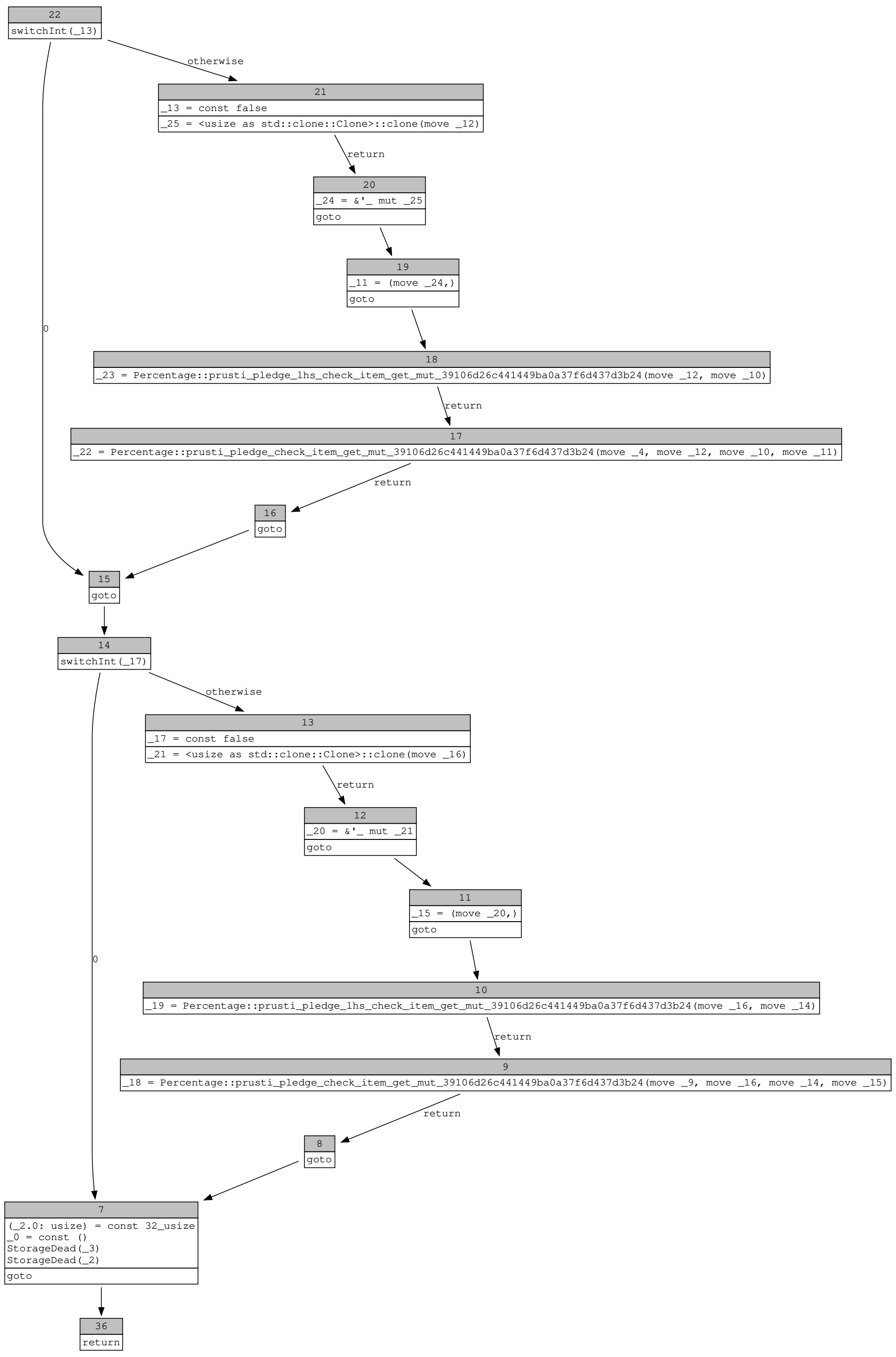
Appendix

A.1 Conditional Pledge MIR Modifications

The modified MIR of example in Listing 4.6, split into two parts.



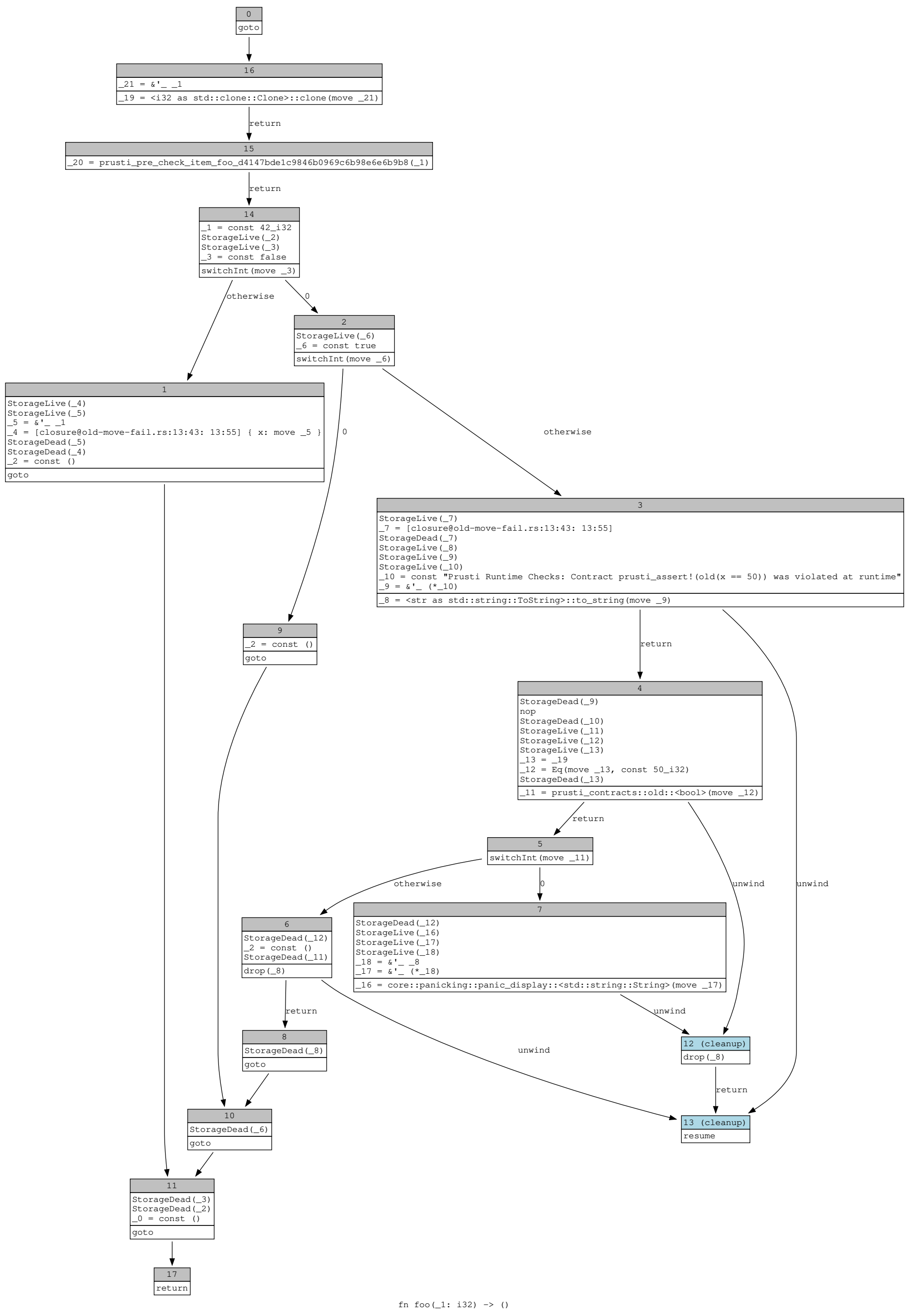
```
0
goto

35
StorageLive(_2)
_2 = Percentage(const 50_usize)
nop
StorageLive(_3)
StorageLive(_4)
_4 = &'_ mut _2
goto

32
_13 = const true
goto

31
_10 = (const (),)
_3 = Percentage::get_mut(move _4)

1
_12 = _3
StorageDead(_4)
nop
StorageLive(_5)
StorageLive(_6)
_6 = _1
switchInt(move _6)

2
goto

4
_5 = const ()
goto

30
switchInt(_13)

29
_13 = const false
_29 = <usize as std::clone::Clone>::clone(move _12)

28
_28 = &'_ mut _29
goto

27
_11 = (move _28,)
goto

26
_27 = Percentage::prusti_pledge_lhs_check_item_get_mut_39106d26c441449ba0a37f6d437d3b24(move _12, move _10)

25
_26 = Percentage::prusti_pledge_check_item_get_mut_39106d26c441449ba0a37f6d437d3b24(move _4, move _12, move _10, move _11)

24
goto

23
(_2.0: usize) = const 43_usize
StorageLive(_7)
StorageLive(_8)
StorageLive(_9)
_9 = &'_ mut _2
goto

34
_17 = const true
goto

33
_14 = (const (),)
_8 = Percentage::get_mut(move _9)

3
_16 = _8
_7 = &'_ mut (*_8)
StorageDead(_9)
_3 = move _7
StorageDead(_7)
StorageDead(_8)
_5 = const ()
goto

6 (cleanup)
resume

5
StorageDead(_6)
StorageDead(_5)
(*_3) = const 62_usize
goto

22
switchInt(_13)
```



```
22
switchInt(_13)

21
_13 = const false
_25 = <usize as std::clone::Clone>::clone(move _12)

20
_24 = &'_ mut _25
goto

19
_11 = (move _24,)
goto

18
_23 = Percentage::prusti_pledge_lhs_check_item_get_mut_39106d26c441449ba0a37f6d437d3b24(move _12, move _10)

17
_22 = Percentage::prusti_pledge_check_item_get_mut_39106d26c441449ba0a37f6d437d3b24(move _4, move _12, move _10, move _11)

16
goto

15
goto

14
switchInt(_17)

13
_17 = const false
_21 = <usize as std::clone::Clone>::clone(move _16)

12
_20 = &'_ mut _21
goto

11
_15 = (move _20,)
goto

10
_19 = Percentage::prusti_pledge_lhs_check_item_get_mut_39106d26c441449ba0a37f6d437d3b24(move _16, move _14)

9
_18 = Percentage::prusti_pledge_check_item_get_mut_39106d26c441449ba0a37f6d437d3b24(move _9, move _16, move _14, move _15)

8
goto

7
(_2.0: usize) = const 32_usize
_0 = const ()
StorageDead(_3)
StorageDead(_2)
goto

36
return
```

A.2 Inline Assertion Translation MIR

The modified MIR of function `foo` in Listing 4.7.



```
0
goto

16
_21 = &'_ _1
_19 = <i32 as std::clone::Clone>::clone(move _21)

15
_20 = prusti_pre_check_item_foo_d4147bde1c9846b0969c6b98e6e6b9b8(_1)

14
_1 = const 42_i32
StorageLive(_2)
StorageLive(_3)
_3 = const false
switchInt(move _3)

2
StorageLive(_6)
_6 = const true
switchInt(move _6)

1
StorageLive(_4)
StorageLive(_5)
_5 = &'_ _1
_4 = [closure@old-move-fail.rs:13:43: 13:55] { x: move _5 }
StorageDead(_5)
StorageDead(_4)
_2 = const ()
goto

3
StorageLive(_7)
_7 = [closure@old-move-fail.rs:13:43: 13:55]
StorageDead(_7)
StorageLive(_8)
StorageLive(_9)
StorageLive(_10)
_10 = const "Prusti Runtime Checks: Contract prusti_assert!(old(x == 50)) was violated at runtime"
_9 = &'_ (*_10)
_8 = <str as std::string::ToString>::to_string(move _9)

9
_2 = const ()
goto

4
StorageDead(_9)
nop
StorageDead(_10)
StorageLive(_11)
StorageLive(_12)
StorageLive(_13)
_13 = _19
_12 = Eq(move _13, const 50_i32)
StorageDead(_13)
_11 = prusti_contracts::old::<bool>(move _12)

5
switchInt(move _11)

6
StorageDead(_12)
_2 = const ()
StorageDead(_11)
drop(_8)

7
StorageDead(_12)
StorageLive(_16)
StorageLive(_17)
StorageLive(_18)
_18 = &'_ _8
_17 = &'_ (*_18)
_16 = core::panicking::panic_display::<std::string::String>(move _17)

8
StorageDead(_8)
goto

12 (cleanup)
drop(_8)

10
StorageDead(_6)
goto

13 (cleanup)
resume

11
StorageDead(_3)
StorageDead(_2)
_0 = const ()
goto

17
return

fn foo(_1: i32) -> ()
```

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, 2019.

[2] Rust contributors. Polonius: The next-generation borrow-checker for rust, 2023. Accessed: 2023-10-18.

[3] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[4] Eva Charlotte Mayer. Assertion-based testing of go programs. Master's thesis, ETH Zurich, Programming Methodology Group, 11 2020. Supervised by Prof. Dr. Alexander Pretschner and Prof. Dr. Peter Müller. Advisors: M.Sc. Linard Arquint and M.Sc. Felix Wolf. Available: `https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Eva_Charlotte_Mayer_MA_report.pdf`.

[5] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[6] The Rust Foundation. The Rust programming language, 2023. Accessed: 2023-10-18.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Contract Checking at Runtime and Verification-based Optimizations for a Rust Verifier

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hegglin	Cedric

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 18.10.2023	C. Hegglin

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.