Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verification of Rust Generics, Typestates, and Traits

Master Thesis

Matthias Erdin

February 13, 2019

Advisors: Prof. Dr. Peter Müller, Vytautas Astrauskas, Federico Poli

Department of Computer Science, ETH Zürich

Abstract

Prusti is the Viper verification infrastructure frontend for the Rust programming language. It allows us to statically guarantee desirable properties of Rust programs, from the absence of unwanted behaviours to the functional correctness according to specifications. The usefulness of verification tools depends on the range of supported language features and idioms. We aim to extend this range and specifically address support for *generics*, *traits*, *invariants* and *typestates*. Generics and traits are the primary mechanism to achieve polymorphism and code-reuse in Rust and are pervasively used. The concept of type invariants is well-known and its usefulness established. The ownership type system of Rust allows expressing and enforcing typestates at compile time, which is useful in security-critical applications. We define semantics and constructs that should be familiar to Rust programmers, and demonstrate their effectiveness in the form of proof-of-work implementations for Prusti.

Acknowledgements

I would like to thank my supervisor Vytautas Astrauskas for the countless hours of support, for the many instructive discussions, and for all the valuable feedback and suggestions I have received. It is most appreciated. I would also like to thank Federico Poli for all the help provided, especially for taking care of issues with the code. I am grateful to Prof. Dr. Peter Müller for the opportunity to work on the topic of verification in the context of Rust; I have learned a lot. Last but not least, I want to thank the entire Programming Methodology group for their feedback and kindness.

Contents

Chapter 1

Introduction

Rust [2] is a new programming language whose type system and ownership model guarantee memory-safety and thread-safety, while supporting low-level programming and providing performance that aims to be competitive with well-known low-level programming languages such as C++ and C.

The *ownership type system* statically prevents otherwise common issues in low-level programming, such as dangling pointers, data races, and other flavors of undefined behaviour, without the need for a garbage collector.

As helpful as the Rust type system is, it cannot prevent every programmer error, let alone statically. Program verification provides the tools and techniques that allow us to check desirable properties of a program beyond the guarantees of the type system statically; from the absence of assertion failures and other kinds of violations of internal assumptions, to the functional correctness of the program, where we show that programs indeed exhibit the specified behaviour.

Rust is a particularly interesting target for verification. Among other reasons, its type system enforces restrictions on *aliasing*, which allows *deriving* essential information needed for verification *from* the Rust program directly. In other programming languages, the burden of providing that information is on the programmer. Because we can derive much verification-relevant information from the Rust program, the barrier to achieve useful verification results is *lowered* significantly for the Rust programmer [8].

We can increase static safety of our programs further by employing *typestates* [10], which are an established concept where objects can be in different (named) states in different program locations; states statically restrict which operations on objects are considered valid. Typestates are useful to express these sets of allowed operations, and have the validity of state transitions checked at compile time. They are often used in security-critical software, like the Rust TLS library rustls [1].

An important rule of the Rust ownership type system is that each memory location has a single owner. This property is checked statically and can be used to *express typestates in the Rust type system* directly [15] such that the compiler then enforces that state transitions are valid and ensures that methods can only be called in appropriate states.

Consider the following example:

```
1  // socket states
2  struct Bound;
3  struct Listening;
4  struct Connected;
5  // ...
6
7  // base structure for any state
8  struct Socket<S> {
9      fd: i32; // normal fields for socket type
10     state: S; // empty state field, does not use memory
11 }
12
13 // functions that operate on sockets in bound state
14 impl Socket<Bound> {
15     fn listen(mut self) -> Socket<Listening> {
16         // ...
17     }
18 }
19
20 // functions that operate on sockets in listening state
21 impl Socket<Listening> {
22     fn accept(&mut self) -> Socket<Connected> {
23         // ...
24     }
25 }
```

Note the difference in the first parameters of the functions; the parameter of `fn listen` is passed *by value*, meaning that `fn listen` *consumes*, i.e. *takes ownership*, of a socket in bound state, and returns a new socket in listening state. This is a common Rust idiom to express state change. On the other hand, `fn accept` expects a reference, and returns a new socket in connected state. Because it does not consume its argument, the passed socket can be said to remain in its original state.

The main motivation for using typestates as shown is to restrict method calls to those appropriate for the type's current state *at compile time*. Our goal is to find and develop convenient means to enable verification of such idiomatic

code. In order to enable support for typestates, we need to look at support for *generics*, *traits*, and *invariants* first.

In this thesis, we design the semantics (where necessary) and the encoding for *generics*, *traits*, *invariants* and finally *typestates*. We implement basic support for each in Prusti [7], the Viper [13] frontend for Rust. Specifically, our contributions encompass the addition of the following items in the form of encoding design and *proof-of-work implementation* in Prusti:

1. We add basic support for generics, which enables verification of functions and methods that are type-parametric.
2. We add support for trait bounds, which enables more useful examples with generics.
3. We enforce the checking of sound substitutability for trait function and trait method implementers.
4. We allow the user to define type invariants for structs.
5. We provide the construct `assert_on_expiry`, with which an *obligation* can be imposed on callers to be fulfilled after the call, which enables safe access to internal representation while upholding invariants.
6. We add basic support for type conditions in type invariants, pre- and postconditions, which allows specifying invariants for *generics-based typestates*.

The outline of the thesis is as follows: Chapter 2 on page 5 introduces background knowledge that is useful for understanding the following chapters. Chapter 3 on page 15 presents the design decisions made, including the chosen semantics for our constructs and the intended Viper encoding. Chapter 4 on page 47 discusses aspects of the implementation including performance statistics, and presents working examples that use the newly added features. Chapter 5 on page 63 finally sums up the results and presents ideas for future work.

Chapter 2

Background

This chapters aims to provide the necessary background knowledge to understand the following chapters. It contains basic concepts; more in-depth and specialized knowledge may only be introduced in later chapters as it becomes necessary.

2.1 Rust

2.1.1 Generics

Rust *generics* are similar to generics in other languages such as Java, Scala, and C#. Generics allow using type parameters as placeholders for some concrete type in the definition of functions, structs and other entities. Examples are `fn max<T: Ord>(v1: T, v2: T) -> T`, a function that returns the larger of two values of some *ordered* type `T`, or `struct Vec<T>` as a dynamic array of some type `T`. As in Java, Scala, and C#, generics are typechecked once at the definition side. At compile time however, Rust *monomorphizes* generics for performance reasons, meaning that for each instantiation with a concrete type, the parameter is replaced with the concrete type, producing separate machine code for each separate concrete type.

2.1.2 Traits

Rust *traits* are comparable to *interfaces* in languages such as C# and Java. Much like interfaces, traits primarily specify methods and their signatures that implementers must provide, which is checked by the compiler. Methods of traits are sometimes accompanied by pre- and postconditions in the documentation, which implementations are expected to uphold. We would like to be able to specify and verify such pre- and postconditions.

2.2 Viper

Viper [13] is the *verification infrastructure for permission-based reasoning*. In a nutshell, it consists of an intermediate verification language based on permission logic, a translator and backends that ultimately use Z3 [9] for verification. It provides an abstraction layer for verification that greatly facilitates implementing *frontends*, program verifiers for specific programming languages.

The Viper language features that are relevant for this thesis are *fields*, *predicates*, *functions*, and *magic wands*. These are introduced in [13].

2.3 Prusti

Prusti [7] is the Rust frontend for Viper. Its purpose is to allow verification of a range of desirable properties of Rust programs, including the absence of *panics* (out-of-bounds accesses, assertion failures, and other kinds of violations of internal assumptions), the absence of integer overflows, and finally the functional correctness of functions and methods.

2.3.1 Type Encoding

Prusti encodes Rust types as Viper predicates. Each distinct Rust type is encoded as a separate Viper predicate; it will be explained what this means in detail. Rust types supported are integral primitive types, structs, enums and tuples, among others. This section explains how type encoding works for integral types, structs, and generic structs. Other types are omitted, as they work in a similar way to how structs and generic structs are encoded.

Types are not encoded *per se*, that is, the encoding of types is not triggered by type definitions in Rust, but as-needed based on their usage in functions or methods.

Primitive Types

Rust *primitive types*, such as `bool`, `u8`, `i32` etc., are encoded utilizing Viper built-in types `Bool` and `Int` as *field types* in the definition of their predicates. With *overflow checking* enabled, the predicates of signed and unsigned integral types include the technical range of the respective types. See Fig. 2.1 on the facing page for a representation of the Rust type `u8`, the unsigned 8-bit integral type (the *byte*).

Viper *fields* are global, and in principle available on every reference. We use accessibility predicates, like `acc(self.val_int)`, to express that the field `val_int` is accessible on the reference `self`. This effectively *models* that any reference that *is* an `u8` *has* the field `val_int` of type `Int`.

```
field val_int: Int

predicate u8(self: Ref) {
    acc(self.val_int) && self.val_int >= 0 && self.val_int <= 255
}
```

Figure 2.1: Encoding of the Rust *primitive type* `u8`, with *overflow checking* enabled in Prusti. The construct `acc(...)` is called an accessibility predicate; it expresses that we have permission to access field `val_int` on the reference `self`. The predicate definition can be interpreted to mean that if `self` *is* an `u8`, then it *has* a field `val_int` of type `Int`, and the integer is in the given range.

Viper integers (type `Int`) are mathematical, *unbounded* integers. The primitive integral types in Rust correspond to the integers provided by hardware, and are bounded. This is modeled by explicitly specifying the range of allowed values in the predicate. Note that the range of representable values for integers is a *technical* limitation, not a logical restriction. This is unlike a user-defined invariant, which *can* be broken and is only expected to hold at certain defined points in the program.

Simple Structs

Structs are encoded by constructing a predicate that is named after the struct. The predicate definition is straightforward: for every field of the Rust struct, the corresponding Viper field accessibility and Viper predicate becomes part the definition, conjoined. Fig. 2.2 shows a simple struct and Fig. 2.3 shows its encoding.

```
struct Foo {
    bar: u8,
    baz: i32,
}
```

Figure 2.2: A simple Rust `struct` with two fields.

```
field enum_0_bar: Ref
field enum_0_baz: Ref

predicate Foo(self: Ref) {
    acc(self.enum_0_bar) && u8(self.enum_0_bar) &&
    acc(self.enum_0_baz) && i32(self.enum_0_baz)
}
```

Figure 2.3: The encoding of the struct shown in Fig. 2.2. Each struct field is encoded with an accessibility predicate and a type predicate.

As a side note, the `enum_0_` prefix of the field names originates from the fact that Rust structs and enums are both represented in the Rust compiler internal datastructures as "ADTs" (algebraic data types), with the distinction that structs have *one variant*, whereas enums may have more than one. Prusti chooses to not make a distinction in the encoding as far as Viper field names are concerned.

Generic Structs and Monomorphisation

Prusti supports *generic structs* (structs with *type parameters*), as long as in the functions and methods under test the generic structs are fully instantiated, that is, used non-generically, in that every type argument provided does *not* contain type parameters (of the function or method under test). An example follows.

Fig. 2.4 shows a *generic* struct that is used by function `test` with two distinct *type arguments*, namely `i32` and `i64`, for *type parameter* `T` of `struct Packet`. Fig. 2.5 on the facing page shows the encoding of types `Packet<i32>` and `Packet<i64>` via *monomorphization* [8]. This process produces separate encodings as if the Rust programmer *had* manually written distinct types `struct Packet__i32` and `struct Packet__i64` with the parameter `T` replaced accordingly in their definitions. Note that in the bodies of the predicates, the types of the `payload` fields differ.

```
1  struct Packet<T> {
2      urgency: u8,
3      payload: T,
4  }
5
6  fn test(p1: Packet<i32>, p2: Packet<i64>) {
7      // ...
8  }
```

Figure 2.4: A *generic* Rust `struct`, used with two different *type arguments* in `fn test`. In `fn test`, the struct is *fully instantiated* in two different ways; the type arguments (`i32` and `i64`) are concrete types. The function under test itself, `fn test`, is *not* parametric; there are no type parameters used in `fn test`.

2.3.2 Pure Functions

Rust functions and methods can be annotated with the `#[pure]` attribute. It marks a function as being *deterministic* and *side-effect-free*. Rust functions and methods *not* marked pure will be translated to Viper *methods*, whereas Rust functions and methods marked as *pure* will be translated to Viper *functions*.

```
field enum_0_urgency: Ref
field enum_0_payload: Ref

predicate Packet__i32(self: Ref) {
    acc(self.enum_0_urgency) && u8(self.enum_0_urgency) &&
    acc(self.enum_0_payload) && i32(self.enum_0_payload)
}

predicate Packet__i64(self: Ref) {
    acc(self.enum_0_urgency) && u8(self.enum_0_urgency) &&
    acc(self.enum_0_payload) && i64(self.enum_0_payload)
}
```

Figure 2.5: The encoding of the struct shown in Fig. 2.4 on the facing page, illustrating the *monomorphization* of generic types.

The encoding as Viper function enables clients of the pure function or pure method to see and reason with its definition.

Fig. 2.6 on the next page illustrates the difference between a pure and a non-pure function. Fig. 2.7 on the following page shows the corresponding encoding – *significantly* simplified. Note that the first assert verifies because clients can reason with the body of the pure function (`fn add_pure`). If the function is not marked pure (`fn add_non_pure`), its behaviour must be specified explicitly, by adding a postcondition (`#[ensures]`) that tells clients how the result can be expected to relate to the arguments.

Fig. 2.8 on page 11 shows an example with a non-generic pure function that is defined on the generic user-defined type `struct Packet`. Fig. 2.9 on page 11 shows the corresponding *illustrative* encoding, showcasing type monomorphization, as explained in section 2.3.1 on page 6. The aspect to be highlighted is that the Viper function in Fig. 2.9 on page 11 has a parameter that is *typed* by the precondition `requires Packet__i16(self)`, which makes the field accesses in the function body possible. Note that the necessary unfoldings have been omitted for readability.

```
1  extern crate prusti_contracts;
2
3  #[pure]
4  fn add_pure(a: i32, b: i32) -> i32 {
5      a + b
6  }
7
8  #[ensures="result == a + b"] // necessary!
9  fn add_non_pure(a: i32, b: i32) -> i32 {
10     a + b
11 }
12
13 fn main() {
14     assert!(add_pure(2, 3) == 5); // verifies!
15     assert!(add_non_pure(2, 3) == 5); // verifies!
16 }
```

Figure 2.6: An example that illustrates the difference between pure and non-pure functions. Annotating a function with the `#[pure]` attribute allows clients to see and reason with its body.

```
1  function add_pure(a: Int, b: Int): Int
2  {
3      a + b
4  }
5
6  method add_non_pure(a: Int, b: Int): Int
7      ensures result == a + b
8  {
9      a + b
10 }
11
12 method main()
13 {
14     assert add_pure(2, 3) == 5 // verifies!
15     assert add_non_pure(2, 3) == 5 // verifies!
16 }
```

Figure 2.7: The *significantly* simplified encoding of the snippet shown in Fig. 2.6. For the sake of readability, overflow checking was not enabled, hence there are no range limitations on the Viper integers.

```rust
struct Packet<T> {
    urgency: u8,
    payload: T,
}

#[pure]
fn is_urgent(p: &Packet<i16>) -> bool {
    p.urgency > 0
}

fn test(p: &mut Packet<i16>) {
    p.urgency = 0;
    assert!(!is_urgent(p)); // holds
}
```

Figure 2.8: A pure function with a user-defined generic type as parameter. Note that the pure function itself is *not* generic.

```
field enum_0_urgency: Ref
field enum_0_payload: Ref
field val_int: Int

predicate u8(self: Ref) {
    acc(self.val_int) && self.val_int >= 0 && self.val_int <= 255
}

predicate i16(self: Ref) {
    acc(self.val_int) && self.val_int >= -32768 && self.val_int <= 32767
}

predicate Packet__i16(self: Ref) {
    acc(self.enum_0_urgency) && u8(self.enum_0_urgency) &&
    acc(self.enum_0_payload) && i16(self.enum_0_payload)
}

function is_urgent(self: Ref) requires Packet__i16(self) {
    self.enum_0_urgency.val_int > 0
}

method test(p: Ref) requires Packet__i16(p) {
    p.enum_0_urgency.val_int := 0
    assert !is_urgent(p) // holds
}
```

Figure 2.9: Shows a nearly self-contained simplified encoding of the code in Fig. 2.8. As always, folding and unfolding is omitted for readability.

2.3.3 References and Lifetimes

Rust guarantees memory safety without garbage collection. In order to provide such guarantees and allow for flexibility in referencing memory, a number of core features are necessary; among them *borrows* and *lifetimes*. This section aims to provide a quick introduction in how references that are passed to a function but cannot be used immediately again, sometimes called *blocked* references, are handled and translated from Rust to Viper.

The situation is best explained on an example. Consider Fig. 2.10 that illustrates a basic case. Function `fn foo` has two parameters, both references to `struct Outer`. The first parameter reference bears lifetime `'a`, the second parameter reference bears lifetime `'b`, and most importantly, the returned reference also bears lifetime `'b`.

```
1  struct Outer { x: i32, inner: Inner }
2  struct Inner { y: i32 }
3
4  fn foo<'a, 'b>(one: &'a mut Outer, two: &'b mut Outer) -> &'b mut Inner {
5      &mut two.inner
6  }
7
8  fn bar(one: &mut Outer, two: &mut Outer) {
9      let two_inner = foo(one, two);
10     one.x = 42;
11     //two.x = 42; // error: two is "borrowed from" (blocked)
12     two_inner.y = 42;
13     two.x = 42;
14 }
```

Figure 2.10: Rust snippet showing `fn foo` that features in its signature references with lifetime annotations (`'a` and `'b`), and the basic semantics of such references; shown in `fn bar`. (This snippet is only valid under the assumption that we have the newer non-lexical-lifetimes feature of Rust available.)

The significance is the following: The linking of the returned reference with the *second* parameter reference, via lifetime, signals to the caller that the returned reference *may alias* the reference that is passed as the second argument (`two`) or refer to a subobject reachable from it. Therefore, in the calling context, it must be ensured that the reference that was passed as the second argument is *blocked* (*inactive*, be inaccessible) while the reference that is returned is *alive*. This ensures the core rule in the Rust type system: That at any time, there cannot exist two *active* (usable) mutable references to the same object.

How this works can be illustrated on the call site, with `fn bar`: We start

with two *active* references `one` and `two` (our function parameters) in line 8. We pass both references to `fn foo` in line 9 to obtain a third reference, stored in the local variable `two_inner`. In line 10, reference `one` is usable (active) immediately again, because it was not blocked; we use it to set a field in the structure.

To understand the error in the following line, we need to look at how the returned reference is used in our function. In line 12, we use it to set a field in the referred-to structure. Let us assume that we have the new "non-lexical-lifetimes" feature of Rust available; then this line 12 marks the *last* usage of the returned reference `two_inner`. The reference is said to be *alive* to line 12; this marks its *lifetime*.

This has then the following consequences: Via the signature of `fn foo`, the lifetime of `two_inner` determines the lifetime of the borrow of `two`: The reference `two` remains *borrowed from* (or *blocked*) until line 12, to ensure that at no point in the execution there exist two (mutable) references that would allow referring to the place `two.inner`.

Because `two` is considered *borrowed* in line 11, we cannot use it at all, and therefore this access would cause a compiler error. Assuming that we are under the "non-lexical-lifetimes" semantics, the reference `two_inner`, and consequently the borrow on `two`, expire at end of line 12. Therefore, the use of `two` succeeds in line 13, because it is then no longer borrowed.

How borrows are translated to Viper is illustrated in Fig. 2.11, in a much simplified way. It shows the signature that would correspond to `fn foo`. (The encoding shown ignores several details of the real encoding. Among others, it does not reflect the fact the references themselves are encoded too, with their own predicates.)

```
1 method foo(one: Ref, two: Ref) returns (res: Ref)
2     requires
3         Outer(one)
4         && Outer(two)
5     ensures
6         Outer(one)
7         && (Inner(res) --* Outer(two))
8         && Inner(res)
```

Figure 2.11: Viper snippet that shows a *simplified* encoding of the signature of `fn foo` from Fig. 2.10 on the facing page; the *blocking/linking* of references is encoded as a *magic wand*.

The important aspect is the following: The precondition plainly states that we expect to gain full access to two `Outer` instances, with the names `one` and `two`. The postcondition states that we return to the caller the full permission to access `one` (line 6); this corresponds to the most common usage of refer-

ences in Rust, where the *borrow* is alive only during the call, and we regain access immediately on return.

The returned reference is also encoded as expected: It communicates that we gain full access to an instance of `Inner` (line 8). The interesting difference however is how the second parameter is encoded in the postcondition: The full permission to use it is returned *conditionally*, expressed with a *magic wand* (line 7).

The *magic wand* codifies the fact that full access to `two` (on the wand right-hand-side) is only possible after we give up access to `res` (the returned reference): This encodes rather directly the restriction that the Rust type system (and the borrow checker) impose on the caller.

How the call site is encoded is not shown. What happens is essentially this: as the returned reference *expires* after line 12, Prusti automatically performs an *apply* of the magic wand, thereby converting our permission to access `two_inner` back into the permission to access `two`, as the rules of the Rust type system dictate.

Note that `fn foo` bears *explicit* lifetime annotations because it has more than one reference argument; this allows specifying *which* of the parameters are linked with the returned reference. If there is only one reference argument (and one reference returned), the lifetimes can be *elided*. Nonetheless, the semantics then are still the same; in particular, all the beforementioned restrictions on call-site and effects for the encoding still apply.

Chapter 3

Design

This chapter discusses semantics and presents the proposed design and encoding to enable support for *generics*, *traits*, *invariants*, and finally *typestates*. It does not address the implementation or the results yet; these are discussed in chapter 4 on page 47.

3.1 Generics

As first step towards reasoning with typestates, we must add support for generics and type parameters. We aim to enable verification of functions and methods that are themselves generic, and that we verify definition-side instead of on instantiation, as generics are type-checked definition-side in Rust.

This section focuses on the core support, namely type parameters on functions and methods; we do not address traits or trait bounds yet; these are discussed in section 3.2 on page 19.

3.1.1 Type Parameters

We have seen in section 2.3.1 on page 6 that generic types are encoded via *monomorphization*. This works as long as *type arguments* do not contain *type parameters*. In this section, we are going to extend the encoding to support this particular case.

Consider for example Fig. 3.1 on the following page, where the generic type `struct Packet<T>` is used by the function `fn increase_urgency<X>`, which is *itself* generic. As was shown in the background chapter, types are encoded as-needed; the need here is however to encode type `struct Packet` where the type argument for the type parameter `T` of `struct Packet` is the *type parameter* `X` of `fn increase_urgency`. The monomorphization fails because the type parameter `X` does not have a defined encoding.

```rust
struct Packet<T> {
    urgency: u8,
    payload: T,
}

fn increase_urgency<X>(p: &mut Packet<X>) {
    if p.urgency < 255 {
        p.urgency += 1
    }
}
```

Figure 3.1: An example where a generic struct is used generically, that is, with the type argument for parameter `T` being a type parameter `X` in the context of usage.

The proposed solution is simple: type parameters shall be considered a special fundamental type, where the encoding of that type is an *abstract predicate* `typaram`. Fig. 3.2 shows the encoding of `struct Packet` where its type argument is the type parameter of `fn increase_urgency`. The abstract predicate communicates that there *may* be state behind the field `payload`, but that its structure and properties are *not known* at this location. From a principal point of view, using a single predicate for any type parameter should suffice and be helpful to avoid generating redundant monomorphizations, especially for pure functions, as we shall see in the next section.

```
field enum_0_flags: Ref
field enum_0_payload: Ref

// abstract (bodyless)!
predicate typaram(self: Ref)

// "monomorphized" encoding of Packet<X>
predicate Packet__typaram(self: Ref) {
    acc(self.enum_0_flags) && u8(self.enum_0_flags) &&
    acc(self.enum_0_payload) && typaram(self.enum_0_payload)
}
```

Figure 3.2: The encoding of the struct shown in Fig. 3.1, illustrating the *monomorphization* of generic types, where the *type argument* of the struct is a *type parameter* in the context of the type's use.

3.1.2 Pure Functions

Pure functions with type parameters need an additional mechanisms for full support. In the background chapter it was shown that Rust functions and methods that are marked *pure* are encoded as Viper functions. Recall that

the Rust type of a function or method parameter is encoded to Viper as a precondition with a type predicate, which then allows access to the fields as defined for the type in question.

If the types of the parameters of pure functions or pure methods contain type parameters, the respective Viper functions will be encoded using the *type parameter monomorphization* shown in the previous section. As long as such pure functions are used from generic contexts, no problem arises. When they are used from specialized contexts however, where the type parameter is substituted with a concrete type, we encounter the issue that the specialized predicate (for a local variable, for example) does not match the typaram-predicate that is used by the generic Viper function.

In order to solve this mismatch in a simple but effective way, we propose to extend monomorphization to pure functions: In the encoding, Viper functions that correspond to pure functions shall be monomorphized analogously as to how it is performed for types and type predicates. An example follows.

Consider Fig. 3.3 on the following page that shows the *generic pure* function `is_urgent`. It evaluates a field of its argument, but that field does not depend on the type parameter at all. Because of type monomorphization, the functions `test1` and `test2` will be encoded using Viper predicates `Packet__i32` and `Packet__i64`, which are distinct in Viper. The pure function `is_urgent` itself can be encoded with the typaram-aware monomorphization introduced before with the predicate `Packet__typaram`, but this would not be useful: The pure function is used in `test1` and `test2`, where the predicates will be different. To make the encoding work, the predicates need to agree. The encoding would therefore need to have monomorphized versions of `is_urgent`. Fig. 3.4 on the next page shows these monomorphized variants along with the test functions; the encoded Viper functions only differ in the predicate that is expected in the precondition.

```
struct Packet<T> {
    urgency: u8,
    payload: T,
}

#[pure]
fn is_urgent<X>(p: &Packet<X>) -> bool {
    p.urgency > 0
}

fn test1(p: &mut Packet<i32>) {
    p.urgency = 0;
    assert!(!is_urgent(p)); // holds
}

fn test2(p: &mut Packet<i64>) {
    p.urgency = 42;
    assert!(is_urgent(p)); // holds
}
```

Figure 3.3: Rust snippet showing `fn is_urgent` that is both *pure* and *type-parametric* (generic). It is used in two specialized contexts (`X == i32` and `X == i64`). The proposed encoding is shown in Fig. 3.4, where the pure function is *monomorphized*.

```
function is_urgent__i32(self: Ref) requires Packet__i32(self) {
    self.enum_0_urgency.val_int > 0
}

function is_urgent__i64(self: Ref) requires Packet__i64(self) {
    self.enum_0_urgency.val_int > 0
}

method test1(p: Ref) requires Packet__i32(p) {
    p.enum_0_urgency.val_int := 0
    assert !is_urgent__i32(p) // holds
}

method test2(p: Ref) requires Packet__i64(p) {
    p.enum_0_urgency.val_int := 0
    assert !is_urgent__i64(p) // holds
}
```

Figure 3.4: Shows the encoding for Fig. 3.3. The pure function is monomorphized to make Viper predicates agree with the local variable `p` in both test functions.

3.2 Traits

We have established basic support for generics in the previous section. Generics can only be fully useful if we have support for traits and trait bounds, because this will allow using methods on generic types. In this section, we look at various missing aspects towards support for traits.

We would like to start with enabling client-side support for trait bounds, so that generics can be more useful. Fig. 3.5 shows a code snippet that we want to support. It turns out that to support this, there is no further encoding or further rules necessary; the existing support for generics is sufficient. The trait bound is transparent for our design. It is of course relevant in the realization: The bound lets us find the definition of the function we call in line 10, which allows us to use the relevant specifications from lines 2–3, in order to make verification work.

```
1  trait Rng {
2      #[requires="low < high"]
3      #[ensures="low <= result && result < high"]
4      fn gen_range(&mut self, low: usize, high: usize) -> usize;
5  }
6
7  #[requires="values.len() > 0"]
8  fn pick<R: Rng>(rng: &mut R, values: &[i32]) -> i32 {
9      // asserts that index in range!
10     values[rng.gen_range(0, values.len())]
11 }
```

Figure 3.5: Rust code snippet showing a simple use of generics with trait bounds. We assume that there is an implicit assert in line 10 that checks whether the supplied index is within bounds, which is guaranteed by the postcondition of `fn gen_range`.

3.2.1 Refinement

Traits are used to specify an interface an implementer must adhere to. It is desirable to uphold the substitutability principle [11] with respect to the specifications of the base trait method and the trait method implementation.

In this section, we are going to look at *specification refinement*. We will look at reasons for and against supporting it in the context of Rust, then different ways to support it, and which of these suits the common cases best.

Motivation and Non-Motivation

The motivation for supporting specification refinement for trait methods is to allow clients that are aware of the implementer (are using the implement-

ing type) can work with more refined specifications than those clients that are unaware (coding against the specifications of the (base) trait method).

Consider Fig. 3.6 that shows an example for specification refinement. We have a trait method `fn produce_answer` that produces an integer result. The (base) trait method *ensures* that the result is within certain loose bounds (-100 to 100). An implementing type, here `struct DeepThought`, might want to provide stronger guarantees, and/or have weaker requirements.

```
1  trait Answer {
2      #[ensures="result >= -100 && result <= 100"]
3      fn produce_answer(&self) -> i32;
4  }
5
6  struct DeepThought { /* ... */ }
7  impl Answer for DeepThought {
8      #[ensures="result == 42"]
9      fn produce_answer(&self) -> i32 { /* ... */ 42 }
10 }
11
12 fn test(dt: &DeepThought) {
13     let answer = dt.produce_answer();
14     // desirable: should hold
15     assert!(answer == 42);
16 }
```

Figure 3.6: Rust code snippet showing the case of trait method specification refinement.

The desired behaviour is that the `assert!(..)` in line 15 holds. We can argue that it is statically known, from the type declarations in `fn test`, that the call to `fn produce_answer` in line 13 dispatches to the implementation defined in line 9, and hence its specifications should be effective in the call location.

Consider `std::io::Write::write` as a more realistic example from the Rust standard library: The (base) trait method would have the following postcondition (simplified):

```
1  #[ensures="result >= 0 && result <= buf.len()"]
```

This indicates that anything from nothing to everything may be written from the buffer by a call to `fn write`. The implementation of this trait method for `std::vec::Vec` (the dynamic array type) however can give stronger guarantees, namely

```
1  #[ensures="result == buf.len()"]
```

because writing to the vector is guaranteed to succeed in full (modulo memory exhaustion, which is handled by a different mechanism in Rust).

Support for specification refinement is not strictly speaking necessary, to achieve the desired effect. In Rust, there exists a workaround that can be built easily: The trait method implementation can *dispatch* manually (call) the inherent method implementation that is defined on the implementing type directly. A client that operates on the implementing type (and not the trait), will call (and be bound to) the inherent method implementation, because it shadows the identically named method from the trait. An example follows.

```
1  trait Answer {
2      #[ensures="result >= -100 && result <= 100"]
3      fn produce_answer(&self) -> i32;
4  }
5
6  struct DeepThought { /* ... */ }
7  impl Answer for DeepThought {
8      // assuming: we cannot refine; therefore just dispatch!
9      fn produce_answer(&self) -> i32 { self.produce_answer() }
10 }
11 impl DeepThought {
12     #[ensures="result == 42"]
13     fn produce_answer(&self) -> i32 { /* ... */ 42 }
14 }
15
16 fn test(dt: &DeepThought) {
17     let answer = dt.produce_answer();
18     // will hold!
19     assert!(answer == 42);
20 }
```

Figure 3.7: Rust code snippet showing the workaround for when trait method specification refinement is not available. This is in contrast to Fig. 3.6 on the preceding page.

Consider Fig. 3.7 that shows how the workaround works. We have introduced an identically named function `fn produce_answer` that is defined on the implementing type directly in line 13. This so-called *inherent* (as opposed to *trait*) implementation now bears the actual code. The trait method implementation *only dispatches* (calls) the inherent implementation. This workaround has the same observable effects as the code shown in Fig. 3.6 on the preceding page, namely that the assert in line 19 holds. The main reason why this works is that both in lines 9 and 17, by Rust lookup rules, the inherent method *shadows* the identically named trait method.

Refinement Designs

For the following discussion of refinement designs, let P and Q be the pre- and postconditions declared and effective on the (base) trait method, let U and V be those *declared* on the trait method implementation, and let X and Y denote the *effective* pre- and postconditions of the trait method implementation. The *declared* ones are given by user-supplied attributes (annotations), whereas the *effective* ones are used for verification (the "contract").

Note that in order for overriding to be sound (obey behavioral subtyping), it must hold that

$$P \implies X$$
$$Y \implies Q$$

which must be established *by construction*, or be checked as necessary, depending on the refinement solution chosen. At least the following designs are possible:

(A) No Refinement The most straightforward design would simply *prohibit* specification refinement. Trait method implementations would always inherit the specifications declared on the (base) trait method:

$$X \equiv P$$
$$Y \equiv Q$$

The advantage is that substitutability does not need to be checked, because it is trivially given since the specifications of base and implementation are identical. The main disadvantage is the lack of flexibility; albeit that might not weigh as heavily, given the arguments made and the workaround presented above.

(B) No Relation The other straightforward design would be to let the effective and declared specifications be identical:

$$X \equiv U$$
$$Y \equiv V$$

The advantage is that this is explicit: Every function and method always bears the specifications that are effective for it; this helps readability and understandability. The main disadvantage is that it does not fit well to the most common use case, namely that the specifications of trait method implementations do not deviate from the ones defined in the trait.

(C) Selective Replacement The designs (A) and (B) can be combined as follows: If any precondition is declared on the trait method implementation (with at least one attribute), then that precondition is effective, otherwise the precondition of the (base) trait method is effective. The same rule would apply identically but separately for the postcondition.

$$\text{If any } U \text{ declared, } X \equiv U, \text{ otherwise } X \equiv P.$$
$$\text{If any } V \text{ declared, } Y \equiv V, \text{ otherwise } Y \equiv Q.$$

This has the advantage that the most common use case is respected (where no specification refinement happens), but the rules still allow refinement where necessary.

(D) Supplementing Another approach would be to ensure that the *effective* specifications are correct by construction. The *declared* precondition on the trait method implementation would be combined with the (base) trait method precondition by disjunction (OR-ed), and postcondition accordingly by conjunction (AND-ed).

$$X \equiv P \vee U$$
$$Y \equiv Q \wedge V$$

The advantage is that these rules make the refinement correct by construction; it would allow refinement but not require checking for correct behavioral subtyping. The disadvantages are that it may be rarely useful; and that the disjunction may cause slowdowns in the verification process.

(E) Conditional Precondition Strengthening There are more possibilities. One is "conditional precondition strengthening"; the pre- and postconditions declared on the trait method implementation would state that *if* callers obey stronger requirements, they gain stronger guarantees:

$$X \equiv P$$
$$Y \equiv Q \wedge (old(U) \implies V)$$

These and similar constructions are interesting, but the main disadvantage is that these are quite specialized; they may not be useful in general. If in specific cases such constructions are desired, they can be specified manually, as long as specification refinement is permitted at all.

Compromise It seems to us that (C) provides a good compromise. We assume that in most cases, a trait method implementation would want inherit the specifications of the (base) trait method, which is what solution (C) does by default. But in some cases, refinement would be desirable, and solution (C) allows for that.

3.2.2 Substitutability

The previous section discussed reasons and ways in which to support specification refinement. In this section we take a step back again, and look at the general case where specifications of the trait method implementation *can differ* from the specifications of the base trait method arbitrarily.

We look at how substitutability checks with respect to specifications can be performed. Special rules and checks that apply for pure functions and pure methods are discussed in the next section.

Let's assume that *BaseRequires* and *BaseEnsures* denote the pre- and postconditions defined on the (base) trait method respectively, while *ImplRequires* and *ImplEnsures* denote the pre- and postconditions defined on the implementing (overriding) method respectively; see Fig. 3.8 for an illustration. As was mentioned before, to ensure that it is valid to substitute the base with the implementer, it must hold that

$$\begin{aligned} \mathit{BaseRequires}(\mathit{args}) &\implies \mathit{ImplRequires}(\mathit{args}) \\ \mathit{ImplEnsures}(\mathit{args}, \mathit{result}) &\implies \mathit{BaseEnsures}(\mathit{args}, \mathit{result}) \end{aligned}$$

```
1  trait Trait {
2      #[requires="BaseRequires(args)"]
3      #[ensures="BaseEnsures(args, result)"]
4      fn method(/* args */) -> /* result */;
5  }
6
7  struct Impl {}
8  impl Trait for Impl {
9      #[requires="ImplRequires(args)"]
10     #[ensures="ImplEnsures(args, result)"]
11     fn method(/* args */) -> /* result */ { /* code */ }
12 }
```

Figure 3.8: Snippet of Rust code showing the basic case for implementing (or overriding) a trait method. The pseudo-functions *BaseRequires*, *BaseEnsures*, *ImplRequires*, *ImplEnsures* are meant as placeholders for the actual expressions.

If these implications were encoded as such, they would bear the limitation that the expressions must not contain Viper permissions or predicates; a Viper restriction. Since in Prusti parameters and returns are encoded with predicates though, it is desirable to perform these checks in a way that allows left-hand-side and right-hand-side of the implications to contain predicates.

The necessary checks can be performed with the (hereby dubbed) "envelope technique"; see Fig. 3.9 on the next page for an illustration. The technique

entails the following: For each trait method implementation that does not trivially conform to the base trait method specification (because it simply inherited them, for example), two methods are generated in the encoding.

```
method Trait__method__Impl(/* args */) returns (/* result */)
    requires ImplRequires(args)
    ensures  ImplEnsures(args, result)
// bodyless

method Trait__method__Impl_(/* args */) returns (/* result */)
    requires BaseRequires(args)
    ensures  BaseEnsures(args, result)
{
    // substitutability check
    result := Trait__method__Impl(args)
}
```

Figure 3.9: The "envelope method" technique, where the refined specifications are checked to conform to the base method specifications. Note that this is pseudo-Viper code; especially the result value would need proper treatment, syntactically.

The first generated method bears the specifications of the trait method implementation (the override). It can be bodyless; its purpose is to serve the substitutability check. The second generated method bears the specifications of the *base* trait method base, and its body *calls* the first generated method, passing arguments, and handling the returned value accordingly.

This is not the only way to check for these implications; but this way is intuitive: It effectively models *dynamic dispatch*, calling the implementation (override) from the base method.

3.2.3 Pure Functions

Recall that marking a function or method as *pure* has two effects: It asserts that the function is deterministic and side-effect-free, and it exposes the definition (the body) of the function or method to the caller to reason with.

The first property is easily dealt with: For a trait function or method, being pure must put the same restrictions on the implementer. In other words, if the (base) trait function or method is marked pure, then, as a rule, the implementing (overriding) function or method *must* be marked pure, too.

The second property is not so obvious to handle; there are some design decisions to be made. There are two cases to be discussed: Functions and methods without a body, henceforth called "required methods", and those with a body (a default implementation), henceforth called "provided meth-

ods"; see Fig. 3.10 for a commonly used example from the Rust standard library.

```
trait PartialEq {
    // required method
    fn eq(&self, other: &Self) -> bool;

    // provided method
    fn ne(&self, other: &Self) -> bool
    { !self.eq(other) }
}
```

Figure 3.10: Simplified version of the PartialEq trait from the Rust standard library.

For required methods, the usual semantics of *pure* cannot apply, since there is no body to be exposed that clients then could use to reason with. A solution could be to encode these functions as *abstract* (bodyless) functions in Viper; so that they can still be used in specifications, albeit only to the extent user-provided specifications provide explicit meaning to them. (The next section will explain this further.)

For provided methods, the usual semantics of pure *can* apply, but it is not immediately clear that they should: provided methods *can* be overridden; if the body of the provided method is exposed, it determines the final behaviour of this method, without allowing implementers to change. In other words, since clients of the provided method *use* the body (the definition), they must be able to trust that the given method, even when overridden, obeys the base method definition. Marking provided methods as pure then effectively 'locks' the definition for implementers.

The Rust standard library uses provided methods abundandtly, and in many cases the definition *does* restrict the implementer. One well-known example is `PartialEq::ne` (see Fig. 3.10), where the interface documentation of `trait PartialEq` [6] explicitly requires implementers of `PartialEq::ne` to obey the semantics shown in the default implementation, namely that it is the strict logical negation of `PartialEq::eq`.

We believe that cases where the default implementation should be usable for clients is more common. We also believe that having a simple and uniform semantics for the `#[pure]` attribute is better for usability and understandability.

We therefore make the following conclusions and design decisions: Trait functions and methods that are marked *pure* have the same semantics as non-trait functions and methods marked so. Concretely it means that these functions are always encoded as Viper functions *with* a body, that is, the

definition of the pure function or pure method is exposed.

Consequently, we want to impose the following restriction: Since *pure* should always imply that the body is available for reasoning, "required methods" *cannot* be marked pure, as they do not have a body. (The next section will provide an alternative.)

Last, we must enforce that implementers (overriders) of pure functions and pure methods do not deviate from the definition in the trait. *In addition* to the substitutability checks that are performed for all trait functions and methods, we must check that the override has the same observable behaviour.

Fig. 3.11 *sketches* how such checking can be encoded. Here we assume that we are looking at `trait PartialEq`, where the implementer has overridden `fn ne` to use three negations instead of one. This is not a very useful example, but it shall demonstrate the principle. Lines 3–5 encode the (authoritative) *base* implementation, lines 7–9 encode the to-be-checked override; lines 11–13 encode the check that these two implementations yield the same result. Of course, both implementations use the *same* definition of function `eq`, as there is only ever one definition active.

```
1  function eq(self: Ref, other: Ref): Bool
2
3  function ne__base(self: Ref, other: Ref): Bool {
4      !eq(self, other)
5  }
6
7  function ne__impl(self: Ref, other: Ref): Bool {
8      !!!eq(self, other)
9  }
10
11 method test(self: Ref, other: Ref) {
12     assert ne__base(self, other) == ne__impl(self, other)
13 }
```

Figure 3.11: A self-contained Viper snippet that attempts to provide a *sketch* of how substitutability checking for pure functions can be encoded.

3.2.4 Abstract Pure

We have established in the previous section that `#[pure]` should always have the same effect: among others, the function or method so marked should have its definition (the body) exposed to clients and usable for reasoning. We have imposed the restriction that `#[pure]` therefore cannot be used for functions or methods that do not have a body (sometimes called "required methods" in traits).

Recall that marking functions or methods *pure* has two main effects: (1) it marks the function as deterministic and side-effect-free (meaning it is encoded as Viper function and can be used in specifications), and (2) it enables clients to reason with the body, by encoding the Rust function body as the Viper function body.

There are cases where only effect (1) is desired, but effect (2) is not. We therefore propose the new attribute `#[abstract_pure]`, a weaker variant of `#[pure]` that differs only in that the function or method definition (its body) is *not* exposed. Functions and methods marked *abstract pure* are encoded to *abstract* (bodyless) Viper functions.

The first use case for `#[abstract_pure]` are trait methods that should be pure but have no default implementation. Albeit one could use `#[pure]` in such cases, and have it *automatically* be encoded abstractly if there is no body, we have decided against these semantics for uniformity and understandability (see previous section).

For example, consider Fig. 3.12 where `trait PartialEq` is now fully annotated. Note that `fn ne` *could* be annotated as abstract pure, but this would change semantics: It would then *not* expose to clients that `fn ne` is the strict negation of `fn eq`.

```
1  trait PartialEq {
2      #[abstract_pure]
3      fn eq(&self, other: &Self) -> bool;
4
5      #[pure]
6      fn ne(&self, other: &Self) -> bool
7      { !self.eq(other) }
8  }
```

Figure 3.12: Simplified version of the PartialEq trait from the Rust standard library, with the recommended usage of annotations.

The second, and rarer, use case for `#[abstract_pure]` are (pure) trait methods that *do* have a default implementation, but that should not expose that default implementation for reasoning to clients, nor restrict overriders to that definition. See Fig. 3.13 on the facing page for an example taken from the Rust standard library. One of the "provided methods" of a standard library iterator is `fn size_hint` whose purpose is to give an *estimate* (lower/upper bounds) of how many elements the iterator is still going to yield. The default implementation is simply the *least specific* estimate (maximally loose bounds); clearly, implementers want to change that definition.

```
trait Iterator {
    /* ... */
    #[abstract_pure]
    fn size_hint(&self) -> (usize, Option<usize>)
    { (0, None) }
    /* ... */
}
```

Figure 3.13: An excerpt from `trait Iterator` from the Rust standard library. Since we may want to use `fn size_hint` in specifications, it should be pure. Because the default implementation however is not an authoritative definition and will be overridden, we annotate as *abstract pure*.

3.3 Invariants

In this section, we will explain how support for *invariants* of Rust *structs* and *enums* can be added. For simplicity, we assume that *visibility* and *privacy* are no concern; it is assumed that all types and fields are accessible from all functions, methods and specifications (all types and fields are *public*).

3.3.1 Basic Support

The concept of type invariants, or *class invariants* [12], is well known within the field of computer science. They specify assertions that are meant to hold for all instances of a particular type, across functions and methods. They encode truths and basic assumptions that are thought to hold "always" about every instance of the type.

We would like to use the attribute `#[invariant]` to let users specify invariants for user-defined types (structs and enums). (This attribute already exists in Prusti, for *loop invariants*, which are otherwise orthogonal to this thesis; using the same attribute for struct and enum invariants does not pose a conflict.) Users may use multiple such annotations; their expressions will be logically conjoined.

We propose that as part of the invariant expression, the *instance* of the struct or enum can be referred to with the `self` keyword. Since in Rust, methods always use `self` to refer to the instance, this will be what Rust programmers expect and understand.

Invariant expressions may refer to `self`, *pure* functions and *pure* methods. Any memory location reachable through `self` and field access may be mentioned.

Examples for invariants can be seen in Fig. 3.14 on the next page for structs, and in Fig. 3.18 on page 33 for an enum. Enums are discussed in more

detail in the next section, where we propose syntactic sugar for expressing invariants for enum variants.

```
#[invariant="self.left < self.right"]
#[invariant="self.top < self.bottom"]
struct Rectangle {
    left: i32,
    right: i32,
    top: i32,
    bottom: i32,
}

// "tuple struct"
#[invariant="self.0 <= self.1"]
struct Range(usize, usize);
```

Figure 3.14: Example snippet showing invariants for the types `struct Rectangle` and `struct Range`. Most structs are regular structs that use fields like *Rectangle*, whereas simpler structs like *Range* are often created as *tuple structs* where the elements are referred to with field access syntax, but with an index starting from zero instead of field names.

Invariants shall be encoded as Viper functions. For generic structs and enums, they are *instantiated* (monomorphized) as needed; the mechanism was shown in the background chapter (section 2.3.1 on page 6) and explained again for generics (section 3.1.2 on page 16).

The body of the Viper functions that represent invariants for structs are constructed as follows: It is the logical conjunction of the user-supplied invariant expressions (the assertions supplied with one or more `#[invariant]` annotations), and the invariant function of every field of the struct.

Invariants are assumed and checked, by default, at call boundaries. In particular, invariants are assumed for function arguments, and checked for return values and function arguments that were passed by mutable reference. This works as long as no reference argument is *blocked*; this case will be discussed in section 3.3.4 on page 35.

We propose to introduce the *valid(place)* syntactical construct to indicate that the invariant of *place* (a variable, the return value, or any field expression thereof) holds. Since invariants are assumed and asserted *by default*, using this construct is normally not necessary. Fig. 3.15 on the facing page illustrates, by explicitly asserting *valid(place)* (redundantly), how the user can think of how invariants relate to pre- and postconditions. Fig. 3.16 on the next page shows the corresponding *much simplified* encoding of the method signature.

Implicitly breaking invariants (will be illustrated with an example shortly)

```
impl SomeType {
    // these valid(...) expressions are redundant here!
    #[requires="valid(self) && valid(arg)"]
    #[ensures="valid(self) && valid(arg) && valid(result)"]
    fn some_method(&mut self, arg: &mut OtherType) -> ThirdType {
        /* ... */
    }
}
```

Figure 3.15: Rust snippet that illustrates with the *valid(place)* construct, which asserts that the invariant of *place* holds, how invariants relate to pre- and postconditions (because the equivalent of these *valid(place)* expressions are inserted automatically by default).

```
method SomeType__some_method(self: Ref, arg: Ref) returns (res: Ref)
    requires SomeType(self) && SomeType__valid(self)
    requires OtherType(arg) && OtherType__valid(arg)
    ensures SomeType(self) && SomeType__valid(self)
    ensures OtherType(arg) && OtherType__valid(arg)
    ensures ThirdType(res) && ThirdType__valid(res)
{ /* ... */ }
```

Figure 3.16: Shows a *much simplified* encoding of Fig. 3.15.

should work without specialized support. The Rust type system enforces the *shared* exclusive-or *mutable* rule. This means in particular the following: While a function or method holds exclusive access to an instance, it is guaranteed by the type system that no other function or method (concurrently or recursively) can *even read* that instance. Conversely, while a function or method holds non-mutable (shared, read-only) access to an instance, it is guaranteed by the type system that no other function or method can *mutate* that instance.

```
#[invariant="self.even % 2 == 0"]
struct Deep { even: i32, /* many other fields, deep structure */ }

fn test(one: &mut Deep, two: &mut Deep) {
    one.even += 1;
    // calling anything on 'two' is okay here (not on 'one' though)
    one.even += 1;
}
```

Figure 3.17: Rust snippet that illustrates the meaning of *implicitly breaking invariants*, and how the strict rules of the Rust type system are helpful here.

What is meant by *implictly breaking invariants* is illustrated with Fig. 3.17. We have `struct Deep`, with one field that is meant to be always *even*, as

expressed with the invariant. This struct may have *arbitrary* fields in addition. In `fn test`, we *implicitly* (without annotation) break the invariant on the argument `one` in line 5. While the invariant of `one` is broken in line 6, we are free to soundly call any function on `two`, that is, we can pass `two` to functions that require the invariants of *all observable* instances (objects) to hold. We restore the invariant of `one` in line 7, and can be sure that the invariants of both `one` and `two` hold in the end, as required.

The above works, because the Rust type system enforces the rule that two *active* (usable) *mutable* references do not alias; and furthermore, that neither points to a subobject of the other. In this case it guarantees that `one` and `two` refer to completely distinct *trees* of objects; using one cannot interfere with the other. As long as we do not pass `one` to another function, we can therefore be sure that its broken invariant cannot be observed anywhere, recursively or concurrently.

3.3.2 Enum Variants

The standard Rust way to write expressions that involve enums is with the Rust-native construct called the *match expression*; see Fig. 3.18 for an example. The consequence is that for enums, one would normally have to write a single invariant annotation that contains one match expression that specifies the assertions for *all* enum variants. This is what is naturally supported by Rust, but that might get unwieldy for enums with a large number of variants.

It is a fact that the different ways in which an enum variant can be expressed (Fig. 3.18) are *analogous* to the different ways in which a struct can be expressed (Fig. 3.14 on page 30). Note that the enumeration contains *variants* `Message::Range` and `Message::Rectangle` that are identical in structure to `struct Range` and `struct Rectangle`, respectively.

Therefore, it might be desirable to allow users to annotate enum variants in the way they would be annotated if they were structs; Fig. 3.19 illustrates the idea. The advantages are clear: increased readability and maintainability.

There are issues with this proposal however: Referring to the variant "instance" with `self`, as if the variant was a type, may be surprising to the Rust programmer, because Rust currently does not allow this natively. Rust supports binding patterns to identifiers with the `identifier @ pattern` syntax [3]. This syntax does not help however: If applied to an enum variant, the bound identifier still refers to the enum instance, not the variant.

The other issue is that it might introduce an ambiguity: it lets Rust programmers use syntax that isn't available in the language in this form. Concretely, since the invariant defined on the enum variant can on the one hand refer to fields of the variant (like `self.left`), but on the other hand also use

```
#[invariant="match self {
    Message::None => true,
    Message::Range(a, b) => a <= b,
    Message::Rectangle { left, right, top, bottom }
        => left < right && top < bottom,
}"]
enum Message {
    None,
    Range(usize, usize),
    // "struct-like enum variant"
    Rectangle { left: i32, right: i32, top: i32, bottom: i32 },
}
```

Figure 3.18: Example snippet showing an invariant for type `enum Message`. The enumeration has three *variants*, `None`, `Range`, and `Rectangle`. These are the three ways (syntaxes) available to declare enum variants; without fields, with unnamed fields (tuple-like), or with named fields (struct-like).

```
enum Message {
    None,
    #[invariant="self.0 <= self.1"]
    Range(usize, usize),
    #[invariant="self.left < self.right"]
    #[invariant="self.top < self.bottom"]
    Rectangle { left: i32, right: i32, top: i32, bottom: i32 },
}
```

Figure 3.19: An enum where the *enum variants* are annotated with invariants, instead of the entire enum with a match expression. Compare with Fig. 3.18 and Fig. 3.14 on page 30.

pure methods defined on the enum (like `self.area()`), there is the corner case of having fields that are callable, where an ambuigity between calling the field, and using regular methods would appear *because* of the syntactic sugar we have added. Luckily, Rust syntax strictly distinguishes between method calls and field calls; Fig. 3.20 on the following page illustrates this.

3.3.3 Inhibition

Once type invariants are automatically assumed and asserted at function boundaries, there must exist the possibility to inhibit this automatic assuming and asserting selectively. Operations on data structures often need to break an invariant temporarily. Such operations may be split across multiple functions and methods for readability and maintainability. Naturally, at the boundaries of these (internal) methods, the type invariant may not hold.

Some proposals [14] use a function-style *broken(obj)* construct as part of pre- or postcondition to indicate that the invariant of *obj* may not hold, in the pre-

```
1  struct Test {
2      f: fn() -> i32,
3  }
4
5  impl Test {
6      #[pure]
7      fn m(&self) -> i32 { /* ... */ }
8
9      fn test(&mut self) {
10         // correct
11         let a = (self.f)();
12         let b = self.m();
13         // invalid syntax
14         let c = self.f(); // error: no method 'f'
15         let d = (self.m)(); // error: no field 'm'
16     }
17 }
```

Figure 3.20: Snippet that illustrates Rust syntax for calling fields versus calling methods: Calling a field (that, for example, stores a function pointer) requires parentheses around the field expression (line 11); omitting the parentheses (line 14) is an error. Conversely, calling a method does not use parentheses (line 12); using parentheses is an error (line 15).

or postcondition, respectively. Since Prusti uses the attributes `#[requires]` and `#[ensures]` for specifying pre- and postconditions, this might be confusing however: Fig. 3.21 illustrates the situation where *broken(obj)* syntax *within* the pre- and postcondition could be so interpreted that these functions *require* or *ensure* the invariant to be broken, which is not the intended meaning.

```
1  #[requires="broken(self)"]
2  fn fixes_the_invariant(&mut self) { /* ... */ }
3
4  #[ensures="broken(self)"]
5  fn can_break_the_invariant(&mut self) { /* ... */ }
```

Figure 3.21: How using the *broken(obj)* construct would look like. This is *not* what we propose. One issue is that this construction, in the context of Rust and Prusti, may be confusing, because we do not strictly speaking *require* or *ensure* that the invariants on `self` do not hold; that would be meaningless.

In addition, it is not clear how a *broken(obj)* construct would be handled and encoded. The problem is that *broken(obj)* does not correspond to any expression in that location; instead, it marks the *absence* of the automatically conjoined *valid(place)* constructs that represent the invariants of the places

mentioned.

For the above reasons, we instead propose a new attribute `#[broken]` that lists *place expressions* (limited to parameter names and `result` for the return value) for which the invariant *is not assumed/asserted* to hold. This combines well with the already introduced *valid(place)* construct that can be used to refer to an invariant. Fig. 3.22 illustrates the proposed syntax (compare with Fig. 3.21 on the preceding page).

```
#[broken="self"]
#[ensures="valid(self)"]
fn fixes_the_invariant(&mut self) { /* ... */ }

#[broken="self"]
#[requires="valid(self)"]
fn breaks_the_invariant(&mut self) { /* ... */ }
```

Figure 3.22: The proposed *broken* attribute, together with the earlier introduced *valid(place)* syntax to refer to invariants.

Specifying that invariants do not hold is mostly useful as an *internal* mechanism between priviledged functions and methods for implementing operations on types. One common case that was already mentioned is where operations are split across multiple *helper* functions or *helper* methods. See Fig. 3.23 on the following page for an example.

3.3.4 Assert On Expiry

Recall that invariants are expected to hold at function and method boundaries. It was shown how this is realized if the capability to access arguments is immediately returned to the caller. We have a special situation in Rust, where a reference that was passed to a function cannot be used immediately on return (is *blocked*). We would like to discuss this situation in particular with respect to invariants. (The necessary background was introcuded in section 2.3.3, and is explained well in further detail in [8].)

Compare the Rust snippet shown in Fig. 3.24 on page 37, and the illustrative encoding of the function signatures in Fig. 3.25 on page 37. We have `struct Range` with a simple invariant defined. As was shown before, to enforce that invariants hold at call boundaries (unless suppressed with `#[broken]`), we conjoin Viper function applications that correspond to the respective types' invariants.

The case for `fn advance` is straightforward, as was introduced earlier: Together with the predicates we consider the invariants (as Viper function

```
1  #[invariant="self.beg <= self.end"]
2  struct Range {
3      beg: usize,
4      end: usize,
5  }
6
7  impl Range {
8      pub fn set_beg(&mut self, beg: usize) {
9          self.beg = beg;
10         // here, the invariant is broken, but fix_end() can be called
11         // with a broken invariant, and promises to fix it
12         self.fix_end();
13         // here we have valid(self)
14     }
15
16     #[broken="self"]
17     #[ensures="valid(self)"]
18     fn fix_end(&mut self) {
19         if self.end < self.beg {
20             self.end = self.beg
21         }
22     }
23 }
```

Figure 3.23: Snippet illustrating the use of the newly introduced *broken* attribute and the *valid* construct.

applications) of the respective types to be part of pre- and postconditions (Fig. 3.25 on the next page, lines 5–7).

We would like to extend the applicability of the same principle, namely that invariants hold at call boundaries, and that this is verified modularly. The case that is not handled yet is `fn end_mut` which simply returns a reference to the field `end` of the `struct Range` instance passed. (This may seem like an artificial example; it is a very common Rust idiom however, for many reasons. Among others, accessing elements of data structures often follows this pattern, albeit usually hidden behind syntactic sugar like *index* access to vectors, for example.)

Recall that the Rust type system (the borrow checker) enforces that there can never exist two *active* references that (directly or indirectly via field accesses) allow manipulating the same memory location. In order to enforce this rule, at the call site of `fn end_mut`, the reference that was passed to the parameter `arg` is *blocked* (unusable, "borrowed from"), until the returned reference *expires* (is no longer used).

```
#[invariant="self.beg <= self.end"]
struct Range {
    beg: usize,
    end: usize,
}

fn advance(arg: &mut Range) {
    if arg.beg < arg.end {
        arg.beg += 1
    }
}

fn end_mut(arg: &mut Range) -> &mut usize {
    &mut arg.end
}
```

Figure 3.24: Rust snippet that contrasts the case of `fn advance`, where the passed reference is *not blocked* (usable by the caller immediately after the call), and `fn end_mut`, where the passed reference is *blocked* until the returned reference *expires*. Note that `fn end_mut` *should not* verify, as it leaks representation that allows clients to break the invariant of `arg`.

```
1  function Range__valid(self: Ref)
2      requires Range(self)
3  { /* ... */ }
4
5  method advance(arg: Ref)
6      requires Range(arg) && Range__valid(arg)
7      ensures  Range(arg) && Range__valid(arg)
8
9  method end_mut(arg: Ref) returns (res: Ref)
10     requires Range(arg) && Range__valid(arg)
11     ensures  usize(res)
12     ensures  (usize(res) && /* ??? */)
13              --* (Range(arg) && Range__valid(arg))
```

Figure 3.25: The encoding that corresponds to Fig. 3.24; showing the signatures of the two functions (simplified).

(As a side note, the linking of the lifetimes of returned references to references passed as arguments usually requires *lifetime annotations*; in simple cases like here, these annotations can be *elided*, but the semantics are the same.)

As was shown in the background chapter (section 2.3.3 on page 12), Prusti expresses the *blocking* of the passed reference in the encoding as a *magic wand*, see Fig. 3.25, lines 12–13. The magic wand roughly means the fol-

lowing, from the caller's perspective: Exhaling the magic wand left hand side (giving up permissions and asserting the conditions), allows inhaling the magic wand right hand side (regaining permissions and assuming the conditions).

Prusti automatically does an *apply* of the magic wand in the calling context (which roughly has the effect of exhaling the left hand side and inhaling the right hand side), in the moment the returned reference *expires*. In that moment, the calling context *regains* access to the full instance of the `struct Range` that was passed as an argument.

Our principle that invariants hold at call boundaries, now extends to this scenario naturally as follows: Upon regaining access to the `struct Range` instance, the invariant of that instance must hold. We express this in the encoding by conjoining the invariant (as a Viper function application) right next to the `Range` predicate that expresses access to the instance (line 13 in Fig. 3.25 on the preceding page) on the right hand side of the magic wand.

To sum up, lines 12–13 now read as follows to the caller: By giving up access to `res` (the returned reference), we regain access to `arg` (the passed reference), with the referenced `struct Range` instance in *valid* state (the invariant holding).

The verifier however now correctly *refuses* to verify `fn end_mut` itself: The postcondition may not hold. The cause is that the function makes a bogus guarantee in its postcondition; it *cannot* in fact guarantee that the invariant of `arg` holds upon expiry of the returned reference. And the reason is clear: We allow the client to mutate the state of that instance (via the returned reference to a field) in ways that *may* cause breakage of the invariant. The verifier catches this.

What is missing is appropriate restriction of how the returned reference can be used – or rather, in what state it is *allowed to expire*. When the returned reference expires, its target (ultimately, the field `arg.end`) must relate to the field `arg.beg` as required by the invariant, for the invariant not to be broken.

In order to allow `fn end_mut` to express this restriction, we propose the construct `assert_on_expiry<`*ref*`>(`*obligation*`[, `*consequence*`]`). The *ref* parameter can be used to specify *which* reference expires. This defaults to `result` (the returned reference, if the return type is a single reference). The angle brackets can then be omitted. The *obligation* is an expression that evaluates to a truth value; it is the *condition*, or the *restriction* we impose on the caller. This condition is *asserted* to hold *on expiry* of *ref* (*right before* expiry of *ref*). (The optional parameter *consequence* allows to relate the state before and after expiry; its purpose is explained later in the section.)

The `assert_on_expiry` construct is valid only as part of the postcondition. We encode the construct by *conjoining the obligation to the left hand side of the*

magic wand; the place indicated with question marks on line 12 in Fig. 3.25 on page 37. With this construct, `fn end_mut` can safely leak representation of its argument, and have the correctness verified modularly. The resulting snippets can be seen in Fig. 3.26 and Fig. 3.27.

```
1 #[invariant="self.beg <= self.end"]
2 struct Range { beg: usize, end: usize }
3
4 #[ensures="assert_on_expiry(old(arg.beg) <= *result)"]
5 fn end_mut(arg: &mut Range) -> &mut usize {
6     &mut arg.end
7 }
```

Figure 3.26: Rust snippet where `fn end_mut` was enhanced with the newly introduced `assert_on_expiry` construct. It puts an *obligation* on the caller the ensure that upon expiry of the returned reference the given condition holds. (Note that with these specifications, we actually require the caller to *establish* the condition, by writing to `*result`; the caller cannot rely on the condition holding without writing to `*result`, because `fn end_mut` does not *ensure* so by postcondition.)

```
1 method end_mut(arg: Ref) returns (res: Ref)
2     requires Range(arg) && Range__valid(arg)
3     ensures  usize(res)
4     ensures  (usize(res) && old(arg.beg.value) <= res.value
5               --* (Range(arg) && Range__valid(arg))
```

Figure 3.27: The encoding that corresponds to Fig. 3.26, where the *obligation* of the `assert_on_expiry` construct was added to the left hand side of the magic wand in line 4. Note that the encoding is simplified, in particular the field accesses and the omitted but necessary *unfolding*.

The terms (`assert_on_expiry`, and *obligation*) were chosen because of the effective semantics of the construct. Because Prusti applies the magic wand *when* the returned reference expires, it then effectively *asserts* the obligation. Hence the caller cannot chose to ignore this obligation; from the caller's perspective, the called function or method *ensures* that the obligation *will be asserted on expiry*. Despite being part of the postcondition, it effectively is a *requirement* on the caller.

It is worth noting that `assert_on_expiry` does *not* require the user (the author of the function or method) to expose the (full) invariant of the arguments. Firstly, the *obligation* only needs to mention the part that is relevant; Fig. 3.28 on the following page illustrates this. Secondly, it not necessary to exactly replicate the relevant parts of the invariant. Clearly, it is only required that the *obligation* is restrictive *enough*; the author of the function or method can impose stronger restrictions than the invariant would require.

```
#[invariant="self.m2 % 2 == 0"]
#[invariant="self.m3 % 3 == 0"]
#[invariant="self.m5 % 5 == 0"]
struct Example { m2: usize, m3: usize, m5: usize }

#[ensures="*result == old(arg.m3)"]
#[ensures="assert_on_expiry(*result % 3 == 0)"]
fn m3_mut(arg: &mut Example) -> &mut usize
    &mut arg.m3
}
```

Figure 3.28: Another example for `assert_on_expiry` that illustrates the following aspect of this construct: It allows leaking representation of a type with invariants safely, and it is not necessary to expose the full invariant.

Lastly, the optional parameter *consequence* of `assert_on_expiry` enables us to relate the state before and after expiry. The *consequence* will be treated as-if it had been specified as the argument to the already existing `after_expiry` construct [8].

3.4 Typestates

So far we have discussed generics, traits and invariants. They are prerequisites for adding support for typestates. In this section, we will first discuss support for generics-based typestates, where states are represented with type arguments, and then discuss support for specification-only typestates, where reasoning with states happens on the verification layer only.

3.4.1 Generics-based

We use *generics-based typestates* to refer to the idiom that uses generics (type parameters and type arguments) to encode the *state* of a type. One common example would be `struct Socket<S>` where the purpose of type parameter `S` is to signal the (underlying) state of the socket. For that parameter, the type arguments `Listening`, `Bound`, `Connected` and so forth can be supplied. These type arguments would usually be *unit structs* (structs with no fields), whose purpose is merely to serve as state indicators.

For a type like `struct Socket<S>`, we want to be able to express the *invariants* dependent on `S`. Consider Fig. 3.29 on the next page. In this example, the use case might be as follows: If `S` is `Closed`, the associated *file descriptor* field `S` shall be `-1`. In all other cases, it shall be `>= 3`, to indicate a *valid* file descriptor. Of course, this is a simple example; more complex and useful cases would involve other states and richer fields that encode addresses, cache information and so forth.

```
struct Closed;
struct Bound;
struct Listening;
// ... more states

struct Socket<S> {
    // file descriptor
    // if S == Closed: fd == -1
    // if S != Closed: fd >= 3
    fd: i32,
    // more fields ...
}
```

Figure 3.29: Motivating example for generics-based typestates, where the invariants are different *depending* on the argument to the type parameter `S`.

In order to allow users to express these kinds of invariants, we propose an extension to the specification grammar. The extension is a new kind of implication, written as `-->` (dash dash angle-bracket), whose right hand side is any assertion, and whose left hand side is a newly introduced *type condition*. Type conditions are simple grammars themselves. They can use conjunctions, disjunctions, and most importantly relations (equality or inequality) of *elementary type expressions*. Type expressions will be parsed with the Rust parser, and should allow specifying types in the same way they can be expressed in the Rust program. Fig. 3.30 shows an overview of the proposed grammar.

```
typestate implication = type condition, "-->", assertion;
type condition = tc disjunction;
tc disjunction = tc conjunction, { "||", tc conjunction };
tc conjunction = tc atom, { "&&", tc atom };
tc atom = tc relation | "(", tc disjunction, ")";
tc relation = type expression, ( "==" | "!=" ), type expression;
type expression = ? type expression parsed via rust compiler ?;
assertion = ? generic assertion as supported in Prusti ?;
```

Figure 3.30: Proposed grammar (EBNF) for *typestate implications* that would become part of the specification grammar (in the same priority as regular implications).

Elementary type expressions are parsed with the Rust parser as Rust types. The encoding of type conditions into Viper expressions is straightforward; all syntactical elements are encoded to the corresponding Viper constructs. Only the encoding of *type expressions* needs to be defined.

Type expressions will be encoded as a function application of a Viper func-

tion that represents the specified Rust type. This *type tag* function will have the return type `Int`. The integer value *represents* the Rust type. Distinct Rust types must be represented by distinct integer values.

Depending on the type represented, we choose the following construction for the *type tag* functions: If the type is a type parameter, the corresponding *type tag* function is *bodyless*. This expresses that a type parameter represents *some unknown type*. If the type does not contain a type parameter, the body of the *type tag* function shall be a unique integer that is distinct for distinct Rust types. (Complex types where type arguments are themselves type parameters are not supported.)

The Rust compiler provides a convenient datum for representing a type with a unique integer: In the Rust compiler, types are *interned*; each distinct type is represented with a unique type object in memory. Whether two types are equal is therefore determined by whether their representations in the Rust compiler have the same memory address; this memory address can serve as the unique integer required for the encoding.

How the typestate-aware specification looks like is shown in Fig. 3.31; the illustration of the corresponding encoding is shown in Fig. 3.32.

```
1  struct Closed; struct Bound; struct Listening; // ... more states
2
3  #[invariant="S == Closed --> fd == -1"]
4  #[invariant="S != Closed --> fd >= 3"]
5  struct Socket<S> {
6      fd: i32,
7  }
8
9  fn generic<S>(sock: &mut Socket<S>) { /* ... */ }
10 fn specific(sock: &mut Socket<Bound>) { /* ... */ }
```

Figure 3.31: Simple example with typestates. The invariants use the newly introduced *typestate implications*, where the left hand side is a *type condition*. Effectively, this specifies different invariants based on the type argument (`Closed` or something else) for the type parameter `S`. The two functions serve to illustrate how the monomorphization in the encoding works; see Fig. 3.32 on the next page.

3.4.2 Specification-Only Typestates

This section *sketches* the design for specification-only typestates, where states are *not* encoded with Rust types, but only used in specifications (the "verification layer"). Note that we only define the desired semantics, but not the intended realization (the encoding) in detail.

The main motivation for supporting specification-only typestates is back-

```
// type tag for type parameter S (abstract/bodyless!)
function typaram_S__tag(): Int

// type tag for struct Closed (with arbitrary unique integer)
function Closed__tag(): Int { 123 }

// type tag for struct Bound (with arbitrary unique integer)
function Bound__tag(): Int { 456 }

// generic socket invariant
function Socket__typaram__invariant(self: Ref)
    requires Socket__typaram(self)
{
    (typaram_S__tag() == Closed__tag() ==> self.fd == -1) &&
    (typaram_S__tag() != Closed__tag() ==> self.fd >= 3)
}

// monomorphized socket invariant (S == Bound)
function Socket__Bound__invariant(self: Ref)
    requires Socket__Bound(self)
{
    (Bound__tag() == Closed__tag() ==> self.fd == -1) &&
    (Bound__tag() != Closed__tag() ==> self.fd >= 3)
}

method generic(self: Ref)
    requires Socket__typaram(self) && Socket__typaram__invariant(self)
    ensures  Socket__typaram(self) && Socket__typaram__invariant(self)
    // ...

method specific(self: Ref)
    requires Socket__Bound(self) && Socket__Bound__invariant(self)
    ensures  Socket__Bound(self) && Socket__Bound__invariant(self)
    // ...
```

Figure 3.32: The illustrative encoding of Fig. 3.31 on the preceding page. Note that for `method generic`, the state is unknown since it uses the non-monomorphized invariant where the left hand sides of both implications *might* be true. For `method specific` however, the state is known since the type parameter tag is replaced with the actual type argument tag in the body of the *monomorphized* invariant.

wards compatibility and ease of use: It allows us to reason with states without having to change Rust code. In addition, generics-based typestates may have drawbacks in that they are verbose and less flexible, because there the states are encoded with the Rust type system.

We would like to be able to specify *states* for user-defined types (structs and enums). We define that every type has a predefined default state, called *valid*. We propose a new `#[state]` attribute for *declaring* further states; the grammar is defined in Fig. 3.33. It is meant to resemble the Rust syntax used to declare traits and their supertraits, for familiarity.

```
state attribute value = state declaration;
state declaration = state, [ set of superstates ];
set of superstates = ":", superstate, { "+" superstate };
state = identifier;
superstate = identifier;
identifier = ? Rust identifier ?;
```

Figure 3.33: Proposed grammar (EBNF) for the value of the newly introduced *state* attribute that can be used to *declare* states for structs and enums.

The following rules shall apply: New states cannot be named "valid", this is the default state that already exists for all structs and enums. States cannot be declared twice. Every state has the implicit superstate *valid*, in addition to the declared superstates, if any. The *directed graph of states* that is formed via the list of superstates must be cycle-free (a DAG).

We propose to extend the grammar of the `#[invariant]` attribute as shown in Fig. 3.34. If the invariant attribute names a set of states, the condition/expression in the invariant will then hold not in all states, but only in the set of states specified, *and their substates*. The *default set of states* is the single predefined state *valid*. The consequence is that invariants that do not name states hold in all states, since every state is (implicitly) a substate of *valid*.

```
new invariant attribute grammar = [ set of states ], expression;
set of states = "[", state, { "+", state }, "]";
state = identifier; (* must name a declared state, or "valid" *)
expression = ? the condition that shall hold in these states ?;
identifier = ? Rust identifier ?;
```

Figure 3.34: Proposed grammar (EBNF) for the value of the earlier introduced *invariant* attribute that can be used to specify invariants for structs and enums. It essentially allows specifying invariant conditions that only hold in a specific *set of states*.

States can be used with function-like syntax in specifications (pre- and post-conditions). States are *nominal*, that is, whether a state holds is *not* interchangeable with the conditions defined for that state. The state can be thought to be represented by a hidden (ghost) field of type `bool`. This requires the following extra rule: Priviledged functions (that have access to private fields of a user-defined type) can *assume* states. If they use the ex-

pression *somestate(someplace)* in their postcondition, is will be *assumed* but *not checked* that the ghost field `someplace.somestate` is effectively set to `true`. Unpriviledged functions cannot do this; they must *derive* states by using priviledged functions. Note that even priviledged functions are required to establish the invariants that are meant to hold in the state that they want to *assume*.

The look and feel of specification-only typestates is shown in Fig. 3.35. It uses the hereby introduced attribute `#[preserves]` that is effectively the combination of `#[requires]` and `#[ensures]`, that is, it specifies conditions that are *preserved* over the function call (hold before and after).

```
#[state="closed"]
#[state="open"]
#[state="bound: open"] // substate of open
#[state="listening: open"] // substate of open
#[state="connected: open"] // substate of open

// holds in state "closed"
#[invariant="[closed] self.fd == -1"]

// holds in states "open", "bound", "listening" and "connected"
#[invariant="[open] self.fd >= 3"]

struct Socket { fd: i32 }

#[requires="bound(self)"]
#[ensures="listening(self)"]
fn listen(&mut self, /* ... */) { /* ... */ }

#[requires="listening(self)"]
#[ensures="listening(self) && connected(result)"]
fn accept(&mut self, /* ... */) -> Socket { /* ... */ }

// with "preserves"
#[preserves="listening(self)"]
#[ensures="connected(result)"]
fn accept(&mut self, /* ... */) -> Socket { /* ... */ }

// recovering state at runtime
#[ensures="result ==> connected(self)"]
fn is_connected(&self) -> bool { /* ... */ }
```

Figure 3.35: Rust snippet that illustrates the looks and semantics of specification-only typestates.

Chapter 4

Evaluation

In this chapter, we discuss what has been successfully implemented and demonstrate some of the features in selected test cases. We also present performance measurements on these test cases.

4.1 Implementation

This section discusses the additions made to Prusti, in the form of proof-of-work implementation. Some of these are demonstrated in section 4.3 on page 50 via self-contained test cases.

Basic support was added in all of the main topics, namely *generics*, *traits*, *invariants* and *typestates*. The full design is not covered however, and in one case, we deviate from the design.

We have added support for type parameters, encoded as abstract predicates, as designed. We deviate from the design in that we do not encode a single predicate that would be used for any type parameter, but we distinguish type parameters *by name*. That is, for distinctly named type parameters, distinct predicates are encoded. This is wasteful, but the reason is practical, as explained in the following paragraphs; the design should have worked.

The existing implementation was not prepared to handle generics easily; the core issue was that Viper predicate names are hard-coded into internal data structures that represent the specifications (pre- and postconditions) of functions and methods. These hard-coded predicate names are used to generate some necessary constructs in the Viper encoding (in particular, the fold/unfold statements).

The hard-coded predicate names then caused conflicts in situations where more-generic functions (having more type parameters) use less-generic functions (having fewer type parameters) with specifications. The predicates of

local variables on the one hand, and the predicates in the specifications of the called functions and methods on the other hand then disagreed, causing the encoding to be erroneous.

In order to solve these disagreements we need to encode type parameter predicates *with the name* of the type parameter. This permits the disagreements between client-side predicates and supplier-side predicates to be resolved by *learning* how they disagree, and then applying the learned transformation to the remaining predicates as necessary, such that more-generic functions using less-generic ones can be processed correctly.

This approach is suboptimal; the learning algorithm constructs case-specific regular expressions at runtime to transform Viper predicate names as necessary. This is fragile, and it is likely the reason why more complicated generics test cases (see below) seem to have some performance issues.

Further, we have implemented traits and trait bounds, as designed. Refinement of specifications (from base trait method to trait method implementation) was *not* implemented. Of course, the proper substitutability checks are still performed, namely that implementers obey the specifications of the functions and methods defined in the trait.

We have implemented the `assert_on_expiry` construct as designed, allowing users to effectively specify the left-hand-side of the magic wand in the postcondition. This construct requires the implementation to report a new kind of error, namely when clients fail to establish the *obligation* specified with the construct. Invariants have been implemented for structs, and are assumed and checked to hold as designed. Only the basic mechanism was implemented, not the additional features (like *broken* and *valid*).

Lastly, we have enabled the reasoning with *generics-based typestates* by allowing users to use *type conditions* in invariants, pre- and postconditions. Not the full proposed grammar of type conditions is supported however, only the elementary *equality* comparison between two type expressions.

4.2 Performance Measurements

This section presents the performance (timing) measurements that have been performed on the example test cases shown in section 4.3 on page 50, and briefly discusses the results.

4.2.1 Methodology

The timing measurements have been performed in the following setup:

- Host system: Intel(R) Core(TM) i7-5600U @ 2.60GHz × 2, regular HDD

- Guest system: Virtualized Ubuntu 18.04.1 LTS with 2GB RAM
- Java version: OpenJDK 10.0.2 2018-07-17
- RustC version: nightly-2018-06-27-x86_64-unknown-linux-gnu
- Prusti configuration: *unoptimized development* build with default options (no extra checks enabled such as overflow checking)

For one measurement, the chosen test was run 16 times; the first measurement was ignored; the statistics were calculated on the remaining 15 measurements. Since the benchmarking of each single test case was made in succession and the first run ignored, disk access is negligible (the data was available in the disk cache). Each measurement started Prusti and Java anew; the timings hence include startup and shutdown overhead, and it can be expected that the JVM was *cold* (the JIT did not optimize).

4.2.2 Results and Discussion

Table 4.1 shows the statistics of the timing measurements performed on the example test cases presented in the following sections.

Overall, the running times are within the range of running times of preexisting test cases [8] (where we do not compare the complexity of the test cases however). Therefore we can say that the implementation does not have a *systemic* performance problem.

One observation that must be made however is that more complex cases of generics, as shown in test case "Generics 2", cause the running times to increase significantly. We think this is the result of the algorithm implemented for generics that solves mismatches between predicates in the internal data structures. This algorithm firstly performs many constructions and applications of regular expressions, and secondly, because type parameters are distinguished by name, it causes many redundant and quite lengthy Viper predicates to be generated, likely contributing to the observable drop in performance.

Table 4.1: Timing Measurements (all numbers in seconds)

Example Test Case	mean	std.dev.	maximum
Generics 1	17.6	0.7	19.0
Generics 2	124.0	9.8	131.7
Traits	20.6	1.3	23.0
Invariants	29.4	1.2	30.9
Assert On Expiry	56.7	1.6	58.5
Typestates	52.2	1.7	55.6

4.3 Example Test Cases

This sections presents *simple* tests for each of the main topics covered (generics, traits, invariants, typestates). The purpose of this display is to show samples of the *elementary* features that were *implemented* in Prusti as part of this thesis.

These tests are part of the test suite; they are self-contained and run successfully, which means that the reported verification errors are *exactly* those indicated with the special comment: `//~ ERROR <message>`.

4.3.1 Generics

This section shows two simple test cases for generics. Note that these test cases are restricted to *generics only* (no trait bounds), and are hence not very realistic or practical.

Test Case "Generics 1"

Remarks on test case "Generics 1" (code follows below):

- Lines 3–7 define a *trusted pure* function that is meant to represent a known fact about the generic type. It has to be declared `#[trusted]` to force the encoding to be an *abstract* (bodyless) function, so that the dummy definition of `true` is ignored and *valid(u)* can be reasoned about as an abstract property of `u`.
- Lines 9–12 define a dummy *reader* that does *not mutate* the instance `u`. It must be declared `#[pure]` because Prusti does not distinguish readonly references from mutable references on non-pure functions yet.
- Lines 14-15 define a dummy *writer* that *can mutate* the instance `u`.
- Lines 17–24 define the test that checks whether we can reason with the abstract property *valid(u)* correctly. The precondition (line 17) assumes the property to hold, which is asserted in line 19. Passing `u` to a non-mutating function (line 20) does not invalidate the property, which is asserted in line 21. Passing `u` to a mutating function (line 22) however *does* correctly invalidate the property, as indicated in the error that the assert in line 23 *might not hold*.

```
1  extern crate prusti_contracts;
2
3  #[pure]
4  #[trusted] // pretend to be abstract (bodyless)
5  fn valid<U>(u: &U) -> bool {
6      true
```

```
}

#[pure]
fn read<U>(u: &U) -> bool {
    true
}

fn write<U>(u: &mut U) {
}

#[requires="valid(u)"]
fn test<U>(u: &mut U) {
    assert!(valid(u));
    read(u);
    assert!(valid(u));
    write(u);
    assert!(valid(u)); //~ ERROR assert!(..) statement might not hold
}

fn main() {}
```

Test Case "Generics 2"

Remarks on test case "Generics 2" (code follows below):

- This is a *stress test*; it uses many differently named type parameters, and uses nested type arguments, where the argument is itself a generic type.
- The logic of the test case is straightforward; its purpose is to establish whether handling type parameters works. It tests in particular whether using more-generic functions (incr1, incr2) from less-generic functions (test1, test2, test3) works as expected.
- This is the longest-running test in the test suite. The low performance is likely due to the algorithm that matches more-generic predicates to less-generic predicates, and the many redundant predicates generated in the encoding.

```
extern crate prusti_contracts;

struct Number<A, B, C> {
    a: A,
    b: B,
    c: C,
}
```

```
8
9  #[ensures="arg.b == old(arg.b) - 1000"]
10 fn decr1<D, E>(arg: &mut Number<D, i32, E>) {
11     arg.b -= 1000;
12 }
13
14 #[ensures="arg.b.b == old(arg.b.b) - 1000"]
15 fn decr2<F, G, H, I>(arg: &mut Number<F, Number<G, i32, H>, I>) {
16     arg.b.b -= 1000;
17 }
18
19 #[requires="arg.a.b == 3000"]
20 #[requires="arg.b.b == 5000"]
21 #[requires="arg.c.b == 7000"]
22 fn test1<X>(arg: &mut Number<Number<i8, i32, u8>,
23                              Number<i16, i32, i64>,
24                              Number<X, i32, usize>>) {
25     decr1(&mut arg.a);
26     decr1(&mut arg.c);
27     assert!(arg.a.b == 2000);
28     assert!(arg.b.b == 5000);
29     assert!(arg.c.b == 6000);
30     decr2(arg);
31     //assert!(arg.a.b == 2000);
32     assert!(arg.b.b == 4000);
33     //assert!(arg.c.b == 6000);
34 }
35
36 #[requires="arg.a.b == 3000"]
37 #[requires="arg.b.b == 5000"]
38 #[requires="arg.c.b == 7000"]
39 fn test2<X, Y>(arg: &mut Number<Number<Y, i32, u8>,
40                                 Number<i16, i32, Y>,
41                                 Number<X, i32, usize>>) {
42     decr1(&mut arg.a);
43     decr1(&mut arg.c);
44     assert!(arg.a.b == 2000);
45     assert!(arg.b.b == 5000);
46     assert!(arg.c.b == 6000);
47     decr2(arg);
48     assert!(arg.a.b == 2000); //~ ERROR assert!(..) statement might not hold
49     assert!(arg.b.b == 4000);
50     //assert!(arg.c.b == 6000);
51 }
```

```
52
53 #[requires="arg.a.b == 3000"]
54 #[requires="arg.b.b == 5000"]
55 #[requires="arg.c.b == 7000"]
56 fn test3<X, Y, Z>(arg: &mut Number<Number<X, i32, Z>,
57                                    Number<i16, i32, Z>,
58                                    Number<Y, i32, Y>>) {
59     decr1(&mut arg.a);
60     decr1(&mut arg.c);
61     assert!(arg.a.b == 2000);
62     assert!(arg.b.b == 5000);
63     assert!(arg.c.b == 6000);
64     decr2(arg);
65     //assert!(arg.a.b == 2000);
66     assert!(arg.b.b == 4000);
67     assert!(arg.c.b == 6000); //~ ERROR assert!(..) statement might not hold
68 }
69
70 fn main() {}
```

4.3.2 Traits

This sections contains one simple test case for traits. This requires *generics* as prerequisite and shows *trait bounds* at work.

Test Case "Traits"

Remarks on test case "Traits" (code follows below):

- Lines 3–9 define a simple trait that has two *bodyless* ("required") methods with specifications. The first (`fn get`) is annotated with a postcondition, the second (`fn set`) is annotated with a precondition.
- Lines 14–16 and 25–27 show that the implementations of `fn get` are checked to conform to the postcondition defined in line 4.
- Lines 17–19 and 28–30 show that the implementations of `fn set` have access to the precondition as defined in line 7.
- Lines 33-41 define two test functions that show that the caller has access to the postcondition of `fn get` that is found via the trait bound `T: Percentage`.
- Lines 43-49 define two test functions that show that the caller is checked to establish the precondition of `fn set` that is found via the trait bound `T: Percentage`.

```rust
extern crate prusti_contracts;

trait Percentage {
    #[ensures="result <= 100"]
    fn get(&self) -> u8;

    #[requires="arg <= 100"]
    fn set(&mut self, arg: u8);
}

struct Fail {}

impl Percentage for Fail {
    fn get(&self) -> u8 { //~ ERROR postcondition might not hold
        101
    }
    fn set(&mut self, arg: u8) {
        assert!(arg <= 99); //~ ERROR assert!(..) statement might not hold
    }
}

struct Pass {}

impl Percentage for Pass {
    fn get(&self) -> u8 {
        100
    }
    fn set(&mut self, arg: u8) {
        assert!(arg <= 100);
    }
}

fn test_get_fail<T: Percentage>(t: &T) {
    let p = t.get();
    assert!(p <= 99); //~ ERROR assert!(..) statement might not hold
}

fn test_get_pass<T: Percentage>(t: &T) {
    let p = t.get();
    assert!(p <= 100);
}

fn test_set_fail<T: Percentage>(t: &mut T) {
    t.set(101); //~ ERROR precondition might not hold
```

```
45 }
46
47 fn test_set_pass<T: Percentage>(t: &mut T) {
48     t.set(100);
49 }
50
51 fn main() {}
```

4.3.3 Invariants

Test Case "Invariants"

Remarks on test case "Invariants" (code follows below):

- Lines 3–6 define a simple struct with an invariant.
- Lines 9–21 define two methods that together demonstrate that it is checked whether the constructed `struct Percentage` instance that is returned by value upholds the invariant.
- Lines 23–41 define three methods that together demonstrate that the invariant of `self` holds at call boundaries, namely that (1) the invariant is available for reasoning within the body of the method and (2) the body is checked to reestablish the invariant in the end.
- Lines 44–54 define two tests that demonstrate that the invariant of `perc` holds after the construction of an instance via `Percentage::new`.
- Lines 56–66 define two tests that demonstrate that `struct Percentage` instances can be constructed locally such that the invariant does *not hold*, but in that case calls to functions (`fn incr`) that expect the invariant to hold do not verify.
- Lines 68–80 define two tests that demonstrate that the defined invariant is correctly assumed to hold after the call to `fn incr`.

```
1  extern crate prusti_contracts;
2
3  #[invariant="self.value <= 100"]
4  struct Percentage {
5      value: u8,
6  }
7
8  impl Percentage {
9      #[requires="value <= 100"]
10     fn new(value: u8) -> Self {
11         Percentage {
12             value: value,
```

```
13          }
14      }
15
16      #[requires="value <= 101"] // mistake
17      fn new_fail(value: u8) -> Self { //~ ERROR postcondition might not hold
18          Percentage {
19              value: value,
20          }
21      }
22
23      fn incr(&mut self) {
24          assert!(self.value <= 100);
25          if self.value < 100 {
26              self.value += 1;
27          }
28      }
29
30      fn incr_fail1(&mut self) {
31          assert!(self.value <= 99); //~ ERROR assert!(..) statement might ...
32          if self.value < 100 {
33              self.value += 1;
34          }
35      }
36
37      fn incr_fail2(&mut self) { //~ ERROR postcondition might not hold
38          if self.value <= 100 { // mistake
39              self.value += 1;
40          }
41      }
42  }
43
44  #[requires="x <= 100"]
45  fn test1(x: u8) {
46      let perc = Percentage::new(x);
47      assert!(perc.value <= 100);
48  }
49
50  #[requires="x <= 100"]
51  fn test1_fail(x: u8) {
52      let perc = Percentage::new(x);
53      assert!(perc.value <= 99); //~ ERROR assert!(..) statement might not hold
54  }
55
56  #[requires="x <= 100"]
```

```
57 fn test2(x: u8) {
58     let mut perc = Percentage { value: x };
59     perc.incr();
60 }
61
62 #[requires="x <= 101"] // mistake
63 fn test2_fail(x: u8) {
64     let mut perc = Percentage { value: x }; // bogus construction
65     perc.incr(); //~ ERROR precondition might not hold
66 }
67
68 #[requires="x <= 100"]
69 fn test3(x: u8) {
70     let mut perc = Percentage { value: x };
71     perc.incr();
72     assert!(perc.value <= 100);
73 }
74
75 #[requires="x <= 100"]
76 fn test3_fail(x: u8) {
77     let mut perc = Percentage { value: x };
78     perc.incr();
79     assert!(perc.value <= 99); //~ ERROR assert!(..) statement might not hold
80 }
81
82 fn main() {}
```

4.3.4 Assert On Expiry

Test Case "Assert On Expiry"

Remarks on test case "Assert On Expiry" (code follows below):

- Lines 3–8 define `struct Example` with a number of fields that are meant to have *independent* invariants; each must be divisible by some number.
- Lines 11–17 define the pure function that expresses the "invariant". We use an explicit function so that it is clear where we assume and assert the invariant in the following functions.
- Lines 19–27 define an accessor function for the field `m3`. The specifications use the construct `assert_on_expiry` to ensure that the invariant of `self` cannot be violated by a caller, even though it receives access to a field that is restricted by the invariant. Line 22 expresses the *obligation* that is imposed on the caller: It must ensure that the target of

the returned reference is divisible by three *on expiry* of the returned reference.

- Lines 29–38 show the same function again, however with the *obligation* in line 32 missing (replaced by `true`). The verifier correctly refuses to verify: Without the obligation, the callers of this function might invalidate the invariant.
- Lines 41–54 define two functions that show that the *obligation* imposed by `m3_mut` is checked correctly. In `fn test_fail`, the incorrect usage of the returned reference would have violated the invariant.

```
1  extern crate prusti_contracts;
2
3  struct Example {
4      m2: u32, // multiple of 2
5      m3: u32, // multiple of 3
6      m5: u32, // multiple of 5
7      m7: u32, // multiple of 7
8  }
9
10 impl Example {
11     #[pure]
12     fn valid(&self) -> bool {
13         self.m2 % 2 == 0 &&
14         self.m3 % 3 == 0 &&
15         self.m5 % 5 == 0 &&
16         self.m7 % 7 == 0
17     }
18
19     #[requires="self.valid()"]
20     #[ensures="*result == old(self.m3)"]
21     #[ensures="assert_on_expiry(
22         *result % 3 == 0,
23         self.valid()
24     )"]
25     fn m3_mut(&mut self) -> &mut u32 {
26         &mut self.m3
27     }
28
29     #[requires="self.valid()"]
30     #[ensures="*result == old(self.m3)"]
31     #[ensures="assert_on_expiry(
32         true,
33         self.valid()
```

```
34      )"]
35      //~ ERROR pledge in the postcondition might not hold
36      fn m3_mut_fail(&mut self) -> &mut u32 {
37          &mut self.m3
38      }
39  }
40
41  #[requires="arg.valid()"]
42  #[ensures="arg.valid()"]
43  fn test(arg: &mut Example) {
44      let m3 = arg.m3_mut();
45      *m3 += 3;
46  }
47
48  #[requires="arg.valid()"]
49  #[ensures="arg.valid()"]
50  fn test_fail(arg: &mut Example) {
51      //~ ERROR pledge condition might not hold on borrow expiry
52      let m3 = arg.m3_mut();
53      *m3 += 5; // mistake
54  }
55
56  fn main() {}
```

4.3.5 Typestates

Test Case "Typestates"

Remarks on test case "Typestates" (code follows below):

- The test case demonstrates typestates on type `struct Int` with two states, *even* and *odd*. It is not necessarily a practical example; it serves to demonstrate the semantics and workings of the typestate implementation.
- Line 3 imports the name `PhantomData` from the standard library. It is used in the definition of `struct Int` to absorb the type parameter that signals the state. This is required because the type parameter *must* be used, it would be a compilation error to not do so. It is otherwise not relevant for the test.
- Lines 5–15 define a stateful type `struct Int`. The states are expressed with `struct Even` and `struct Odd`, which are used in the invariants with *type conditions* to restrict the field `i` to be *even* or *odd*, respectively, according to the state. The trait serves to demonstrate how trait bounds can be used to make the purpose of the type parameter

clear. The trait and the trait bounds are not necessary however; the test would work without them.

- Line 17 onwards define methods on the *generic* type `struct Int`, that is, on ints that may be in either state. The use of a differently named type parameter shows that the names of type parameters are only locally meaningful; and that the type conditions utilize the locally defined name.
- Lines 18–34 demonstrate that under some circumstances, regular specifications (here, preconditions) must use type conditions to be able to treat the type generically. Depending on whether the client constructs an int in state *even* or *odd*, the requirement on the argument will be different. The fail case demonstrates that the preconditions are indeed necessary.
- Lines 36–64 define various tests that demonstrate that the state-specific invariants are indeed checked.
- Lines 67–81 define tests that demonstrate that in contexts of concrete usage of the type, where the typestate is known, reasoning with the state-specific invariant is possible, whereas in contexts of generic usage, where the state is unknown, it is not.
- Lines 83–101 define testes that demonstrate that preconditions with type conditions of `fn new` work as expected; that depending on which state is requested, the required condition (even or odd) is correctly checked.

```
1  extern crate prusti_contracts;
2
3  use std::marker::PhantomData;
4
5  trait IntState {}
6
7  struct Even; impl IntState for Even {}
8  struct Odd;  impl IntState for Odd {}
9
10 #[invariant="S == Even --> self.i % 2 == 0"]
11 #[invariant="S == Odd  --> self.i % 2 != 0"]
12 struct Int<S: IntState> {
13     i: i32,
14     s: PhantomData<S>,
15 }
16
17 impl<Z: IntState> Int<Z> {
18     #[requires="Z == Even --> i % 2 == 0"]
```

```
19      #[requires="Z == Odd  --> i % 2 != 0"]
20      fn new(i: i32) -> Int<Z> {
21          Int {
22              i,
23              s: PhantomData,
24          }
25      }
26
27      //#[requires="Z == Even --> i % 2 == 0"]
28      //#[requires="Z == Odd  --> i % 2 != 0"]
29      fn new_fail(i: i32) -> Int<Z> { //~ ERROR postcondition might not hold
30          Int {
31              i,
32              s: PhantomData,
33          }
34      }
35
36      fn test_incr2(&mut self) {
37          self.i += 2;
38      }
39
40      fn test_incr3(&mut self) { //~ ERROR postcondition might not hold
41          self.i += 3;
42      }
43
44      fn test_plus2(self) -> Self {
45          Int {
46              i: self.i + 2,
47              s: PhantomData,
48          }
49      }
50
51      fn test_plus3(self) -> Self { //~ ERROR postcondition might not hold
52          Int {
53              i: self.i + 3,
54              s: PhantomData,
55          }
56      }
57
58      fn test_double(self) -> Int<Even> {
59          Int::new(self.i * 2)
60      }
61
62      fn test_triple(self) -> Int<Even> {
```

```
63          Int::new(self.i * 3) //~ ERROR precondition might not hold
64      }
65  }
66
67  fn test1(int: &mut Int<Even>) {
68      assert!(int.i % 2 == 0);
69  }
70
71  fn test1_fail<S: IntState>(int: &mut Int<S>) {
72      assert!(int.i % 2 == 0); //~ ERROR assert!(..) statement might not hold
73  }
74
75  fn test2(int: &mut Int<Odd>) {
76      assert!(int.i % 2 != 0);
77  }
78
79  fn test2_fail<S: IntState>(int: &mut Int<S>) {
80      assert!(int.i % 2 != 0); //~ ERROR assert!(..) statement might not hold
81  }
82
83  #[requires="i % 2 == 0"] // even
84  fn test3(i: i32) -> Int<Even> {
85      Int::new(i)
86  }
87
88  #[requires="i % 2 == 0"] // even
89  fn test3_fail(i: i32) -> Int<Odd> { // wrong return type state
90      Int::new(i) //~ ERROR precondition might not hold
91  }
92
93  #[requires="i % 2 != 0"] // odd
94  fn test4(i: i32) -> Int<Odd> {
95      Int::new(i)
96  }
97
98  #[requires="i % 2 != 0"] // odd
99  fn test4_fail(i: i32) -> Int<Even> {
100     Int::new(i) //~ ERROR precondition might not hold
101 }
102
103 fn main() {}
```

Chapter 5

Conclusion

We have set out to address the topics of Rust *generics* and *traits*, to support *invariants* and to finally enable reasoning with *typestates*, in the context of Prusti [7], the Viper [13] frontend for the Rust programming language.

We have designed semantics and encoding for *typestates*, and as necessary prerequisites we have also addressed *generics*, *traits*, and *invariants*. We have extended Prusti with basic support for the prerequisities and for *generics-based typestates*, an important Rust programming idiom.

We have presented selected tests from the test suite that demonstrate the correct functioning of the implemented features, and we have shown timing measurements for each presented test case. We can conclude that even for the longest-running test, the performance is acceptable.

5.1 Future Work

This section presents possible future work.

5.1.1 Trait Invariants

Rust traits do not have fields, but can be *stateful* from the perspective of the clients, as state can be communicated via methods. There may be a case for allowing *invariants* to be defined on traits. Necessarily, these invariants would need to be expressed via *abstract pure* functions, directly or indirectly. Method implementations would then be checked to uphold the invariant, by using the implementer's definitions of the abstract pure functions in the trait. An example is shown in Fig. 5.1 on the next page.

```rust
#[invariant="self.len() <= self.capacity()"]
trait LimitedVector<T> {

    #[abstract_pure]
    fn len(&self) -> usize;

    #[abstract_pure]
    fn capacity(&self) -> usize;

    #[pure]
    fn available(&self) -> usize {
        // needs the invariant to pass the overflow check
        self.capacity() - self.len()
    }

    #[requires="self.available() > 0"]
    fn push(&mut self, item: T);
}
```

Figure 5.1: Rust snippet that shows how a trait with an *invariant* could look like.

5.1.2 Specifications for Intrinsic Properties of Traits

We have added support for trait method pre- and postconditions. In Rust, traits are however also commonly used to mark types with *intrinsic properties* [4] or to impose specific requirements that cannot be captured by method pre- and postconditions. As an example, the Rust standard library provides the trait `PartialEq` [6]:

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

The library reference however also defines requirements expressed as invariants on this trait that implementations must obey, specified as follows [6]:

> *Formally, the equality must be (for all* `a`, `b` *and* `c`*):*
>
> - *symmetric:* `a == b` *implies* `b == a`*; and*
> - *transitive:* `a == b` *and* `b == c` *implies* `a == c`*.*

Moreover, the standard library provides a so-called *marker trait* `Eq` [5], derived from `PartialEq`, that does not define further methods at all, but imposes additional requirements only [5]:

> *[...] in addition to* `a == b` *and* `a != b` *being strict inverses, the equality must be (for all* `a`, `b` *and* `c`*):*
>
> - *reflexive:* `a == a`*;*
> - *symmetric:* `a == b` *implies* `b == a`*; and*
> - *transitive:* `a == b` *and* `b == c` *implies* `a == c`*.*

Note well that the invariants imposed by the `PartialEq` and `Eq` traits are *hyperproperties*.

The aim would be to group use cases into classes and find solutions to some of these classes. Examples of such classes of properties are:

- `PartialEq` and `Eq`, whose invariants are hyperproperties
- `Sync` and `Send`, which are related to concurrency and safety
- `Copy`, which is related to optimization and receives special treatment by the compiler.

Bibliography

[1] A modern TLS library in Rust. https://github.com/ctz/rustls. Accessed on 2018-08-31.

[2] Rust progamming language. https://www.rust-lang.org/.

[3] The Rust reference: Identifier patterns. https://doc.rust-lang.org/reference/patterns.html#identifier-patterns.

[4] The Rust standard library: Module std::marker. https://doc.rust-lang.org/std/marker/index.html. Accessed on 2018-08-31.

[5] The Rust standard library: Trait std::cmp::Eq. https://doc.rust-lang.org/std/cmp/trait.Eq.html. Accessed on 2018-08-31.

[6] The Rust standard library: Trait std::cmp::PartialEq. https://doc.rust-lang.org/std/cmp/trait.PartialEq.html. Accessed on 2018-08-31.

[7] A Viper front-end for Rust. http://www.pm.inf.ethz.ch/research/prusti.html.

[8] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. Technical report, ETH Zurich, 2018.

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

[10] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 465–490, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[12] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[13] 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.

[14] A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.

[15] David Teller. Typestates in Rust. https://yoric.github.io/post/rust-typestate/.

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):

Verification of Rust Generics, Typestates, and Traits

Authored by (in block letters):

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

Name(s):	**First name(s):**
Erdin	Matthias

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, 13.02.2019	

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