Formal Specification and Verification Techniques for Mutable References and Advanced Aliasing in Rust

by

David Ewert

B.Sc., University of British Columbia, 2021

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

December 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Formal Specification and Verification Techniques for Mutable References and Advanced Aliasing in Rust

submitted by **David Ewert** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Alexander J. Summers, Associate Professor, Computer Science, UBC
Supervisor

William J. Bowman, Assistant Professor, Computer Science, UBC
Supervisory Committee Member

Abstract

Verification tools allow developers to write specifications (or rules) that are used to verify the correctness of their programs, i.e., that the programs follow the specified rules. Prusti and Creusot are two verification tools that can be used to verify programs written in Rust. The way Rust handles aliasing mutation has interesting implications when it comes to verification. Prusti and Creusot each use different techniques to take advantage of this; Prusti uses separation logic while Creusot uses its prophecy model.

In this thesis, we present several enhancements to these tools that relate to aliasing mutation. The first is a type system for defining well-typed Prusti specifications, which we use to develop a translation from Prusti's specification language to Creusot's specification language. Because of the differences in the way Prusti and Creusot handle mutable references, this translation is non-trivial. We then propose an updated soundness proof for Creusot that allows proving the soundness of a new version of its ghost type. The old version of the ghost type was found to be unsound. The existing soundness proof for Creusot has limitations, which require us to make non-trivial changes to its prophecy model to allow it to prove the soundness of our new version of this ghost type. Lastly, we introduce a new type for Rust that can be used with verification tools such as Prusti or Creusot to verify safety in cases where, because of the aliasing restrictions on Rust's reference types, raw pointers are required to implement data structures.

Lay Summary

When writing programs, developers would like to ensure that the programs they write behave the way they expect. One way to do this is to use formal verification tools that allow developers to write rules or specifications, which are then used to verify their programs behave correctly. One of the more challenging aspects of verification is handling program values that can change. Prusti and Creusot are verification tools that can be used to verify programs written in the Rust programming language. They each use different techniques to take advantage of Rust's safety restrictions about when values are allowed to change. In this thesis, we introduce some extensions to these verification tools, including a translation of Prusti rules to Creusot rules, an updated proof for Creusot, and a new data type for Rust that allows developers to verify that programs that work around Rust's safety restrictions are still safe.

Preface

This thesis is an original work by the author, David Ewert. Some text and diagrams have been adapted from a workshop proposal and presentation describing the same work, authored by David Ewert and Alexander J. Summers [6].

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Terms used in this thesis.

ADT Algebraic Data Type

API Application Programming Interface

SMT Satisfiability Modulo Theories

ZST Zero-Sized Type

Acknowledgments

First and foremost, I would like to thank God for “I can do all things through Him who gives me strength” (Phil 4:13)

I would also like to express my deepest appreciation for my supervisor, Alex Summers, for his continual supervision and valuable feedback. He has provided me with insightful guidance on my research and was instrumental in supporting me in writing my thesis. Without his guidance and persistent support, I would never have been able to complete this thesis. In addition, I would also like to thank William Bowman for being the second reader and for providing his feedback.

Ron Garcia also deserves my gratitude as the first professor who sparked my interest in programming languages. As well, Len Pelletier (UBC CPSC alumni) deserves my thanks; he was my favourite high school teacher, who in grades 9-12 helped develop my love for computer science.

To my immediate and extended family, I owe an immeasurable debt of gratitude. My dad was instrumental in helping me get my thoughts out of my head and my family’s unwavering support, encouragement, and belief in my abilities has helped shape the success of my academic endeavours.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien.

Natural Sciences and Engineering Research Council of Canada
Conseil de recherches en sciences naturelles et en génie du Canada
Canada

Chapter 1

Background and Introduction

1.1 Rust

Rust is a statically typed, general-purpose programming language. It can guarantee memory safety even though it does not rely on garbage collection. In this section, we will highlight some Rust features that are relevant to the rest of this thesis.

Memory safety in Rust means that Rust does not allow dereferencing or freeing a pointer that has already been freed or is uninitialized. Rust also does not allow memory to be accessed from one thread when another thread is writing to the memory unless both operations are "atomic", which prevents reading corrupted data in memory. While by default, Rust is memory safe, it does allow developers to write programs that it cannot guarantee are memory safe, by using the `unsafe` keyword. Marking something as unsafe does not mean is not safe, but shifts the responsibility of ensuring safety from the compiler onto the developer.

Rust has a concept of ownership, where variables are said to *own* memory on the heap. Variables that own memory on the heap can automatically free this memory when they go out of scope and their destructors are run. To prevent accessing this memory after it has been freed, variables of types that own memory on the heap, are not allowed to be reused after they have been passed into a function, because the ownership of the memory is transferred into the function, which consumes the variable. Rust does, however, have a built-in feature that allows variables of some types to be re-used after being passed into a function by having their type

```
fn use_pair(p: (u32, bool)) {
  p.0 += 1;
}

fn test() {
  let x: (u32, bool) = (5, true);
  use_pair(x);
  use_pair(x);
}
```

Figure 1.1: Copy Example

implement `Copy`. When passing variables of these `Copy` types into a function, instead of being consumed, a copy is made that the function can use while leaving the original unchanged; since it is a copy, the caller still owns the original value and the function owns the copy. Types can only implement `Copy` if they do not indirectly own memory, and their fields all implement `Copy`. In Rust, int types, float types, `bool` and `char` are all defined to implement `Copy` and can be reused. In Figure 1.1, `x` is a pair of a `u32` and a `bool` and therefore, implements `Copy` and is allowed to be re-used. Note, however, that even though `use_pair` changes the parameter inside of its call since it is a copy, it does not affect the original `x` which remains `(5, true)`.

In Rust, a `Box<T>` is a type that represents a pointer that points to a `T` on the heap, i.e., it is used when a developer wants to store something on the heap. A `Box<T>` is considered to own this heap memory, so the memory is freed when it goes out of scope, and so `Box<T>` does not implement `Copy`. In Figure 1.2, trying to reuse `y`, which is a `Box<u32>` causes a compiler error since it does not implement `Copy`.

Rust includes *reference* types, which allow temporary transfer of access. Rust has two types of references: shared references which are generally read-only, and mutable references which allow the underlying data to be mutated. Creating a reference to a variable in Rust is referred to as *borrowing*, i.e., borrowing a variable creates a reference to it that can be passed into a function. A variable that has been borrowed is allowed to be used again, but doing this invalidates the reference.

Shared references, `&T`, are created by using the `&` operator on a variable of type

```rust
fn use_box(b: Box<u32>) {
  // b goes out of scope so its memory is freed
}

fn test() {
  let y = Box::new(5);
  use_box(y);
  use_box(y); // ERROR
}
```

Figure 1.2: Not Copy Example

```rust
1 fn use_box(b: Box<u32>) {}
2
3 fn use_shared_box(s: &Box<u32>) {}
4
5 fn test() {
6   let y: Box<u32> = Box::new(5);
7   let sy: &Box<u32> = &y;
8   use_shared_box(sy);
9   use_shared_box(sy);
10  use_box(y);
11  use_shared_box(sy); // ERROR
12 }
```

Figure 1.3: Shared Reference Example

`T`. They implement `Copy`, and can therefore be used several times, but they cannot be mutated. Given a shared reference to type `T` that implements `Copy`, the shared reference can be de-referenced to a `T`. A shared reference becomes invalidated when the variable it was borrowed from gets used or goes out of scope, and after that point, the shared references can no longer be used.

Figure 1.3 shows an example involving shared references. On line 7 the variable `y` is borrowed to create the shared reference `sy`. Lines 8 and 9 both call `use_shared_box` with `sy`, which is allowed since `sy` is a shared reference, which implements `Copy`. Line 10 passes `y` into a function, which invalidates `sy`. Line 11 tries to use `sy` again, but since it has been invalidated this causes a compile error.

```
fn use_mut_pair(p: &mut (u32, bool)) {
  p.0 += 1;
}

fn test() {
  let mut x: (u32, bool) = (5, true);
  let mut mx: &mut (u32, bool) = &mut x;
  use_mut_pair(mx);
  let y: u32 = x.0;
}
```

Figure 1.4: Mutable Reference Example

Mutable references `&mut T` are created by using the `&mut` operator and can be mutated. The change is reflected back in the variable it was borrowed from. They do not implement `Copy`, but the variable that was borrowed to create one can be re-used. Re-using this variable, however, invalidates the mutable reference, so the reference can no longer be used after this point. A mutable reference can be converted into a shared reference, but this consumes the mutable reference similarly to as if it were passed into a function.

In Figure 1.4 `x` is mutably borrowed to create `mx` which is then passed into the `use_mut_pair` function. The changes made to the mutable reference in `use_mut_pair` are reflected back to `x` when we use it again, so `y` ends up set to `6`.

Both shared and mutable references can be *projected* to create references to their inner parts. Some examples include: a reference to a `Box<T>` can be projected to a reference of a `T`, a reference to a struct can project to a reference of one of its fields, a reference to a list can be projected to a reference of one its elements, a reference to a mutable reference to a `T` can be projected to a reference to a `T`. Shared references get projected into shared references and mutable references get projected into mutable references.

Additionally, references have a *lifetime* associated with them. This lifetime is used to indicate when a reference is invalidated; references that have the same lifetime get invalidated at the same, which is known as the *expiry* of the lifetime. When a variable is borrowed, the created reference is assigned a lifetime specific

```
1  fn box_deref<'a>(m: &'a Box<u32>) -> &'a u32 {
2    Box::deref(m)
3  }
4
5  fn use_box(b: Box<u32>) {}
6
7  fn test() {
8    let mut b: Box<u32> = Box::new(5);
9    let mut sb: &Box<u32> = &b;
10   let mut s: &u32 = box_deref(sb);
11   let n1: u32 = *s;
12   use_box(b);
13   let n2: u32 = *s; // ERROR
14 }
```

Figure 1.5: Lifetimes Example

to the borrow. When the variable is used again or goes out of scope, this lifetime expires. Projected references can be thought of as having the same lifetime as the reference they were created from. When writing functions that have references in their signature, the lifetimes of these references are available as lifetime parameters (like type parameters). When calling these functions, these lifetime parameters are often inferred.

Figure 1.5 is an example that uses lifetimes. The signature of the `box_deref` function (on line 1) uses the lifetime parameter `'a` to say that it takes in a shared reference to a `Box<u32>` and returns a shared reference to a `u32` that has the same lifetime. The implementation (on line 2) calls a built-in Rust function that projects the argument to create a reference with the same lifetime as required by the signature. On line 9 in the implementation of `test`, the variable `b` is borrowed to create shared reference `sb`. Line 10 calls `box_deref` on `sb` to produce `s`, the signature of `box_deref` forces `s` to have the same lifetime as `sb`. Line 11 dereferences `s` which is allowed since `u32` implements `Copy`. Line 12 passes `b` into a function which causes the lifetime of `sb` to expire. Line 13 tries to dereference `s` again, but this is not allowed since `s`'s lifetime (which is the same as `sb`) has now expired, which causes a compile error.

In some cases, a developer may want to pass in a mutable reference into a func-

```rust
fn bx_deref_mut<'a>(x: &'a mut Box<u32>) -> &'a mut u32 {
    Box::deref_mut(x)
}

fn test() {
    let mut b = Box::new(5);
    let mut mb = &mut b;
    let m = box_deref_mut(mb);
    let m2 = box_deref_mut(mb); // ERROR?
    *m = 0; // ERROR
}
```

```rust
fn bx_deref_mut<'a>(x: &'a mut Box<u32>) -> &'a mut u32 {
    Box::deref_mut(x)
}

fn test() {
    let mut b = Box::new(5);
    let mut mb = &mut b;
    let m = bx_deref_mut(<&mut _>::deref_mut(&mut mb));
    let m2 = bx_deref_mut(<&mut _>::deref_mut(&mut mb));
    *m = 0; // ERROR
}
```

Figure 1.6: Reborrowing Example

tion, and then use it again afterwards. This does not seem possible, since mutable references do not implement `Copy`. As a workaround the developer could borrow the mutable reference, creating a mutable reference to the mutable reference. The developer would then project this "double" mutable reference to create a single mutable reference with the lifetime of the outer one, and pass this into the function instead. This workaround allows the original mutable reference to be used again after the call (although reusing the original mutable reference would invalidate the mutable reference passed into the function). Since these cases are so common, Rust uses this workaround implicitly whenever a mutable reference variable is passed into a function; this implicit workaround is known as *reborrowing*.

The top half of Figure 1.6 shows an example that relies on reborrowing to compile. Line 9 appears invalid since it reuses `mb` even though it does not implement `Copy`. The bottom half of the figure shows the same example with the rebor-

rowing made explicit. Since this version never passes `mb` directly into a function and only ever borrows it, it is easier to see why line 9 does not have any errors. Line 10 does cause a compile error since using `mb` on line 9 invalidated `m`. Note: `<&mut>::deref_mut` is a built-in function that has the following signature:

```
deref_mut<'a, 'b, T>(x: &'a mut &'b mut T) -> &'a mut T;
```

Although Rust has a concept of subtyping, Rust's subtyping applies exclusively to the lifetimes of references. A reference is a subtype of another reference if its lifetime *outlives* the other's lifetime (i.e., it expires later). When trying to create a lifetime for a common supertype, Rust can combine two lifetimes to create a lifetime that is at least as short as either of them. This combined lifetime gets invalidated if either of its input lifetimes is invalidated.

In the following example:

```
1 let x = 0;
2 let mut y = 1;
3 let sx = &x;
4 let sy = &y;
5 let s = if true { sx } else { sy }
6 y = 0;
7 *s; // ERROR
```

`s`'s lifetime is the combined lifetime of `sx`'s and `sy`'s lifetimes. This makes `s`'s type a supertype of `sx`'s and `sy`'s types which allows the `if` expression to type check. When `y` is used on line 6, it causes the lifetime of `sy` to expire, and since `s`'s lifetime contains `sy`'s lifetime it also expires. Trying to dereference `s` on line 7 causes a compile error since `s` has expired.

Based on the rules around references, ownership, and lifetimes, Rust provides some guarantees about when and what can be changed. For example, Rust guarantees that writing to one mutable reference that was passed into a function will not affect any other parameters of the function (i.e., the mutable reference does not alias any other parameters). Similarly, Rust guarantees that the value of a shared reference will never change. In Rust, these guarantees are known as aliasing XOR mutability[1]. From the developer's perspective, these guarantees make understand-

[1] Rust has certain types (e.g. `Cell<T>`) that opt out of this aliasing XOR mutability. Rust allows shared references to these types to be mutated.

```
1 #[requires(d != 0)]
2 #[ensures(result * d <= n)]
3 #[ensures((result + 1) * d > n]
4 fn div(n: u32, d: u32) -> u32 { return n / d; }
```

Figure 1.7: Specifications Example

ing what can and cannot change easier and help minimize surprising side effects. These guarantees also help with compiler optimizations.

1.2 Verification

Modular verification tools allow developers to write specifications that describe the expected behaviour of a function, and then the tool verifies that the implementation meets this expected behaviour for all inputs. The two main types of specifications are preconditions and postconditions. Preconditions describe properties of the function's parameters that must be true in order to call the function. Postconditions describe properties of the function's parameters and returned values that will always be true after calling the function (as long as the preconditions were met). For both preconditions and postconditions, the specifications are written as boolean-like expressions, typically in a language that closely resembles the one being verified. Figure 1.7 is an example of a function written in Rust that has some specifications. The function is a simple function that does integer division. Line 1 is a precondition stating that `d` must never be 0. Lines 2 and 3 are postconditions that state that the result is the largest integer that is less than or equal to `n` when this integer is multiplied by `d`.

To allow for code reuse within specifications, many verification tools allow certain user-defined functions to be used inside of specifications. These functions must be deterministic and not have observable side effects so that their meaning inside specifications is well defined. These specification functions are sometimes referred to as pure functions (e.g. in Prusti) or logic functions (e.g. in Creusot).

At a high level, many verification tools take a program with specifications and convert it into a format that can be solved by a Satisfiability Modulo Theories (SMT) solver. An SMT solver is a tool which takes in a boolean expression with

some variables and attempts to output whether this boolean expression can ever return true for any possible instances of those variables. The verification tool works by translating the implementation and specifications of each function in a program, to a boolean expression that contains variables (the variables might model, for example, the arguments to the functions). This boolean expression returns true if and only if the specifications do not hold. The boolean expression is then given to the SMT solver, to solve. If the SMT solver returns false, that means that its input can never return true, and in this case, it means that the specifications can never fail, so they must always hold, and therefore the function is verified. If the SMT solver returns "unknown" (i.e., cannot determine if the boolean expression can ever return true) then the verification tool conservatively decides not to verify the function.

If a verification tool verifies a function but it is actually possible for the function to return a value that does not satisfy the postcondition (even when its arguments satisfy the precondition), the verification tool is said to be unsound. A common way to demonstrate the unsoundness of a verification tool is to have it try to verify a function where the precondition is `true` and the postcondition is `false`, in this case, the tool is unsound if it verifies the function and the function returns at all [2].

1.3 Rust Verification

When formally verifying a programming language, an important aspect to consider is how the language handles mutation. In more imperative languages like C, aliasing mutation is allowed, where writing to one pointer could affect the result of reading from another seemingly unrelated pointer. For example, in the following:

```
bool test(int* x, int* y) {
    int a = *x;
    *y = 0;
    int b = *x;
    return a == b;
}
```

The function seems as though it should always return true, but if both x and y point to the same place, that is, they alias each other, then the function could return

[2] If a function never returns it is always considered to satisfy its postcondition.

false. This ability to allow aliasing mutation poses a challenge for verification tools since writing to a pointer could have arbitrary effects on the results of other operations. For example, in the following code, we want to verify that we do not divide by zero. We could add a precondition that requires that *x is not 0, but this is insufficient since there is no guarantee that x does not alias a global variable that gets set to 0 inside of function f.

```
// requires (*x != 0)
int test_div(int* x) {
    int n = f();
    return n / *x;
}
```

To deal with this, verification tools for these types of languages often use a technique called separation logic in their specification language which allows users to specify which functions may write to which pointers and which pointers might alias.

In functional languages like Haskell, mutation is not allowed and functions do not have side effects. These characteristics make verification of simple functions easier since expressions produce the same result independently of when they are run so the tool does not need to worry about which states to evaluate an expression in.

Rust falls somewhere in between by allowing mutation but strictly controlling aliasing mutation. As previously discussed, Rust has an aliasing XOR mutability principle. This principle eases some of the mutation issues we have when verifying languages like C because Rust guarantees a reference argument will not alias with any other accessible mutable reference, which allows verification tools to know that a reference will not be modified by function calls that do not involve this reference.

For example in the following:

```
fn test(x: &i32, y: &mut i32) -> bool{
  let a: i32 = *x;
  *y = 0;
  let b: i32 = *x;
  a == b
}
```

The function will always return `true` since `x` is a reference parameter and `y` is an accessible mutable reference, so Rust ensures that they never alias each other.

As another example, in the following:

```
#[requires(*x != 0)]
fn test_div(x: &i32) -> i32 {
  let n: i32 = f();
  n / *x
}
```

The division operation is safe since `x` is a reference parameter, so Rust ensures `x` does not alias with any accessible mutable references, which implies that `x` could not have been mutated via an aliasing global variable in `f`.

Despite Rust's aliasing XOR mutability principle, Rust does still allow mutation, by using mutable references. Therefore, postconditions still need to be able to describe the side effects that functions can have, and evaluating expressions can potentially still give different results in different states. The implications of verifying programs that use mutable references are shown throughout the rest of this thesis.

In Chapter 2 we discuss two verification tools (Prusti and Creusot) that each have different ways of writing specifications for Rust functions that use mutable references, and we propose a translation between them. Translating Prusti specifications for functions that do not use mutable references is fairly straightforward, but directly translating into Creusot is challenging when the specifications involve mutable references. Instead of translating directly, we first define a type system for Prusti's specifications in order to guide the translation. One of the more complicated aspects of the type system involves checking that dereferences of mutable references are well defined, and in doing this, the type checker provides insight into how Prusti's specification for mutable references should be translated. We then use this type system to define a translation from well-typed Prusti specifications to Creusot specifications including some examples of this translation.

The way Creusot reasons about mutable references involves considering their current value, and the value they will have when they expire. In Chapter 3 we describe how this reasoning, combined with Creusot's implementation of a ghost type, led to an unsoundness. A version of Creusot has been proven sound in

RustHornBelt, but this version does not include the ghost type. We modify this soundness proof so that we can now prove the soundness of most of the features that Creusot's original ghost type had.

Rust's aliasing XOR mutability discipline can be too strict when trying to implement certain kinds of data structures that use mutable references. Rust includes a raw pointer type that does not have any aliasing restrictions and can be freely mutated, but is generally unsafe to use. In Chapter 4 we introduce a new Rust type called GhostPtrToken which allows raw pointer to be used in a way that allows verification tools to verify them for safety. We then describe the Application Programming Interface (API) of the GhostPtrToken type and provide examples of how it can be used.

Chapter 2

Prusti to Creusot Specification Translation

2.1 Introduction

There are several existing tools that allow specifying and verifying the correctness of Rust programs, for example, Prusti [1], Creusot [4], and Verus [11], that each have their own specification language. Being able to translate between specification languages can have several advantages. Having a translation can make it easier to compare the various tools, and also make having them inter-operate easier (verifying a program in one tool that uses a library that was verified in a different tool). In this chapter, we define a translation from Prusti's specification language to Creusot's specification language. The specification languages are fairly similar to each other as well as to Rust itself but diverge particularly in how they handle mutable references. Specifying Rust functions involving mutable references is more complicated since these functions can have observable side effects (by mutating through the reference). For example, in Figure 2.1 the `increment` function's specifications should reflect the fact that the function increments the data stored under the mutable reference `x`. In Figure 2.2 the `get_mut` function projects a mutable reference to a pair. Although the function itself does not have any side effects, the specifications for the function should reflect the fact that modifications made to the returned reference will affect the first element of the pair, and the

```
fn increment<'a>(x: &'a mut u32) { *x += 1; }
```

Figure 2.1: `increment` function

```
fn get_mut<'a>(x: &'a mut (u32, u32)) -> &mut u32 {
  return &mut x.0;
}
```

Figure 2.2: `get_mut` function

second element can no longer be changed.

In this section, we compare the specification languages used by Prusti and Creusot, focusing on their handling of mutable references. Verus is not included because it currently only has limited support for mutable references.

2.1.1 Prusti

Prusti [1] supports mutable references by having the value of an expression in a specification depend on the time (or program state) in which the expression should be considered to be evaluated. Prusti's preconditions and postconditions are considered to be evaluated in the state just before (prestate) and after (poststate) the call, respectively. Prusti also supports *pledge* specifications (written as `after_expiry`), which are considered to be evaluated at the expiry point of the lifetime of the return value, and `old` expressions which force their bodies to be evaluated in the prestate regardless of the surrounding context. In the future, Prusti plans to generalize this notion of pledges to allow the use of arbitrary lifetimes instead of just the lifetime of the return value. Prusti could also generalize `old` expressions to expressions that are interpreted as if they were evaluated in an arbitrarily specified state instead of the prestate specifically. Similar ideas are introduced in Gorse's Master's thesis [7].

Prusti is being actively developed and the semantics of the specification language have changed since the original version, but for the purposes of this thesis, we only consider the original specification semantics.

Figure 2.3 shows the `increment` function from Figure 2.1 but this time

```
#[requires(*x < u32::MAX)]
#[ensures(*x == old(*x) + 1)]
#[ensures(*old(x) == *x)]
// #[ensures(*old(x) == old(*x))]
fn increment<'a>(x: &'a mut u32) { *x += 1; }
```

Figure 2.3: `increment` function with Prusti specifications

```
#[ensures(old((*x).0) == *result)]
#[after_expiry((*x).0 == *result)]
#[after_expiry((*x).1 == old((*x).1))]
fn get_mut<'a>(x: &'a mut (u32, u32)) -> &'a mut u32 {
  return &mut x.0;
}
```

Figure 2.4: `get_mut` function with Prusti specifications

including Prusti specifications. The Prusti specification on line 2 describes that dereferencing `x` in the poststate has a value one larger than dereferencing `x` in the prestate. The specification on line 3 shows that wrapping a variable in `old` does not affect the variable's value, `old` only affects dereferences. Interestingly some newer versions of Prusti treat `old` differently and would not be able to verify line 3, instead, these versions would verify line 4 which does not match the original semantics we present here.

Figure 2.4[1] shows the `get_mut` function from Figure 2.2 but this time including Prusti specifications. The Prusti specifications describe the values of the fields of dereferencing `x` when `'a` expires. Field 0 has the same value as dereferencing the returned value when `'a` expires. Field 1 has the same value as it did in the prestate.

One thing to note with Prusti's specifications is that, as in Rust, expressions are not designed to be well-defined in all states. For example, in Figure 2.5, the specification attempts to describe dereferencing `x` in the poststate when `x` has already been freed. Being able to easily check that Prusti expressions are well-defined is therefore important; we return to this point later.

[1] We often include explicit dereferences for clarity even though Rust could infer them.

```
#[ensures(*x == result)]
fn unbox(x: Box<u32>) -> u32 {
  return Box::into_inner(x);
}
```

Figure 2.5: Ill-defined Prusti specification

```
#[requires(*x < u32::MAX)]
#[ensures(^x == *x + 1)]
fn increment<'a>(x: &'a mut u32) { *x += 1; }
```

Figure 2.6: `increment` function with Creusot specifications

2.1.2 Creusot

Creusot's specification expressions [4] always evaluate to the same value instead of being considered to run in a specific state. Operations that depend on the state, namely dereferencing, implicitly choose states in which they are well-defined. For boxes, there is only one valid state. For shared references, dereferencing in any valid state produces the same value so the choice does not matter. For mutable references, there are two valid states so Creusot has two types of dereference operators which allow users to choose which of the states to dereference in. The `*` (*current*) operator dereferences in the reference's initial state, and the `^` (*final*) operator dereferences in the state where the reference will expire. Figure 2.6 shows the `increment` function from Figure 2.1 but this time including Creusot specifications. The Creusot specifications describe that the value stored in `x` when `x` expires is one larger than `x`'s initial value. Figure 2.7 shows the `get_mut` function from Figure 2.2 but this time including Creusot specifications. The Creusot specifications describe the values of the fields of `x` when `x` expires. Field 0 has the initial value of the return value. Field 1 has the same value as field 1 of `x`'s initial value. The `^` operator is considered to be a function from `&mut T` to `T` so `^` can only be applied to mutable references, for example on line 3 `^(x.1)` would not type check since `(x.1)` implicitly dereferences `x` and has type `u32` which is not a mutable reference.

```
#[ensures((*x).0 == *result)]
#[ensures((^x).0 == ^result)]
#[ensures((^x).1 == (*x).1)]
fn get_mut<'a>(x: &'a mut (u32, u32)) -> &'a mut u32 {
  return &mut x.0;
}
```

Figure 2.7: `get_mut` function with Creusot specifications

In this chapter, we define a translation from well-defined Prusti specifications to Creusot specifications. Initially, we consider defining this notion of well-definedness by having Rust type check the specifications, but then we show that this does not work well for mutable references. To overcome this, we create our own type system to define this notion of well-defined Prusti specifications and use our type system to guide the translation from Prusti specifications to Creusot specifications.

2.2 Prusti Specifications as Run-time Checks

Prusti [1] aims for its specifications to have the meaning of running the expression in the relevant state. A direct interpretation of this would be to view Rust specifications as a description of a theoretical run-time check that could be inserted into the program. The precondition would be evaluated and checked before calling the function and the postcondition afterwards. For example, the following Prusti specification:

```
#[requires(x < u32::MAX)]
#[ensures(result == x + 1)]
fn add1(x: u32) -> u32 { x + 1 }
```

could be thought of as the following run-time checks around each call to the function:

```
fn checked_add1(x: u32) -> u32 {
  assert!(x < u32::MAX);
  let result = add1(x);
  assert!(result == x + 1);
  result
}
```

```
#[requires(*x < u32::MAX)]
#[ensures(*x == old(*x) + 1)]
fn increment(x: &mut u32) { *x += 1 }
```

```
fn checked_increment(x: &mut u32) {
  assert!(*x < u32::MAX);
  let old_1 = *x;
  let result = increment(x);
  assert!(*x == old_1 + 1);
  result
}
```

Figure 2.8: Run-time interpretation of an `old` expression

This concept can be useful for converting specifications into actual run-time checks and can also help specifications to be understood by tools like MirChecker/MIRAI [12], and Kani [14] that use `assert!` statements as their means of specifications. Additionally, this concept can help programmers more easily understand the specifications, by visualizing the code that is theoretically run. Ideally, the theoretical run-time interpretation could be compiled and run. This would allow our notion of well-defined Prusti specifications to be those that Rust can successfully type/borrow-check after they have been converted to run-time checks.

With this run-time checking interpretation in mind, `old` expressions could be understood as `let` binding the inner expression to a variable before the call and then using the variable afterwards[2], e.g. Figure 2.8. Pledges could be understood as a closure that is called at the expiry of the appropriate lifetime, e.g. Figure 2.9.

2.2.1 Issues with Mutable References

Unfortunately, this run-time interpretation does not work well when dealing with mutable references that are used in pledges. In Figure 2.9, `x` is mutably reborrowed on line 13; `x` is then used in lines 15-16 as part of creating the pledge closure, which forces the borrow to end. On line 17 `result` is used as the return value, even though `results`'s lifetime comes from the reborrow of `x` that has already

[2]This understanding of `old` expression does not work when the `old` expression is used in the arm of a `if`/`match` expression.

```
#[ensures(old((*x).0) == *result)]
#[after_expiry((*x).0 == *result)]
#[after_expiry((*x).1 == old((*x).1))]
fn get_first_mut<'a>(x: &'a mut (u32, u32))
  -> &'a mut u32 {
  return &mut x.0;
}

fn get_first_mut_checked<'a>(x: &'a mut (u32, u32))
  -> &'a mut u32 {
  let old_0 = (*x).0;
  let old_1 = (*x).1;
  let result = get_first_mut(x);
  assert!(old_0 == *result);
  run_at::<'a>(|| assert!((*x).0 == *result));
  run_at::<'a>(|| assert!((*x).1 == old_1));
  result
}
```

Figure 2.9: Attempted run-time interpretation of a pledge. `run_at` is a hypothetical function that runs its parameter in the state where its lifetime parameter expires. This does not compile; see Section 2.2.1

ended, and this would cause a compilation error. Another separate error is that on line 15 `result` is reborrowed for the lifetime `'a` since `result` is used in the pledge closure, and this prevents `result` from being returned.

A similar problem arises when a mutable reference is contained inside another type even if the specification does not include pledges. In Figure 2.10, on line 2, `w` is moved into the call to `set_wrap`. The parameter `w` cannot be reborrowed since `w` is not a bare mutable reference, but also cannot be copied since `w` contains a mutable reference and mutable references do not implement `Copy`. On line 3, when the postcondition needs to be checked, `w` has already been moved so `w` cannot be used again which would lead to another compilation error.

Using this direct run-time interpretation would seriously limit the specifications that can be made for these types of functions since they cannot describe the changes made to the mutable reference. If we wanted to work around these restrictions and still have run-time checkable specifications, we would need the Rust type checker to pretend mutable references implemented `Copy`. This is because we

```rust
struct Wrapper<'a>{x: &'a mut u32};

#[ensures(*w.x == new_val)]
fn set_wrap(w: Wrapper<'_>, new_val: u32) -> u32 {
  *w.x = new_val
}
```

```rust
fn checked_set_wrap(w: Wrapper<'_>, new_val: u32) {
  let result = set_wrap(w, new_val);
  assert!(*w.x == new_val); // error
  result
}
```

Figure 2.10: Attempted run-time interpretation of a mutable reference contained in another type

need to have two aliases of the mutable reference: one to pass into the call, and one to keep and check later. The specifications could also be interpreted as unsafe Rust by casting relevant mutable references to raw pointers whenever they need to be copied, as seen in Figure 2.11 and Figure 2.12. In these examples, the raw pointer dereferences are safe since the raw pointers came from mutable references that are expiring as the dereference happens which guarantees they are valid.[3] However, since raw pointer dereferences can cause undefined behaviour, relying on running Rust's type checker on this modified version of the run-time interpretation to determine whether a Prusti specification is well-defined would not work. Instead, the specification language would need to have separate checks to guard against this undefined behaviour.

[3]The raw pointer dereferences could violate Rust's aliasing model so the no-alias optimizations would need to be disabled. Since we do not end up using the direct run-time interpretation this will not be a problem.

```
#[ensures(old((*x).0) == *result)]
#[after_expiry((*x).0 == *result)]
#[after_expiry((*x).1 == old((*x).1))]
fn get_first_mut<'a>(x: &'a mut (u32, u32))
  -> &'a mut u32 {
  return &mut x.0;
}

fn get_first_mut_checked<'a>(x: &'a mut (u32, u32))
  -> &'a mut u32 {
  let old_0 = (*x).0;
  let old_1 = (*x).1;
  let raw_x = x as *const _;
  let result = get_first_mut(x);
  assert!(old_0 == *result);
  let raw_result = result as *const _;
  run_at::<'a>(|| assert!(unsafe{*raw_x}.0
                              == unsafe{*raw_result}));
  run_at::<'a>(|| assert!(unsafe{*raw_x}.1 == old_1));
  result
}
```

Figure 2.11: Revised run-time interpretation of a pledge

```
struct Wrapper<'a>{x: &'a mut u32};

#[ensures(*old(w.x) == new_val)]
fn set_wrap(w: Wrapper<'_>, new_val: u32) -> u32 {
  *w.x = new_val
}
```

```
fn checked_set_wrap(w: Wrapper<'_>, new_val: u32) {
  let old_1 = w.x as *const _;
  let result = set_wrap(w, new_val);
  assert!(unsafe{*old_1} == new_val);
  result
}
```

Figure 2.12: Revised run-time interpretation of a mutable reference contained in another type

2.3 Type System for Specifications

As previously discussed, a direct run-time checking semantics of Prusti's specification language would need to support holding on to aliases of mutable references and then dereferencing them later. Since doing this can cause undefined behaviour we would need to prevent the dereferences in some cases. We would need to track the lifetime of the mutable reference that was cast to a raw pointer and require that the raw pointer is only dereferenced in the state where that lifetime expires. In that state, permission to the pointer is being returned so the raw pointer must be valid. However, since this requirement is based on the exact expiry point of a lifetime, the specification language could not use the lifetime outlives relation for sub-typing like Rust, otherwise we could dereference a mutable reference while the mutable reference is still blocked. In this section, we introduce an alternative approach for defining the well-definedness conditions on specifications, that moves away from the direct run-time interpretation. We instead introduce the syntax and rules for a novel type system that meets these dereferencing requirements and provides us with a definition of well-definedness for Prusti specifications.

2.3.1 Introduction and Syntax

This Prusti specification type system is modelled on the Rust type system, with some modifications. The syntax, shown in Figure 2.13, is similar to Rust's syntax but has a few changes. Our type system includes a notion of *states*, that can be thought of either as a point in time or the program state, i.e., the heap at that point in time. Specifically, only the prestate 'pre, poststate 'post, and the state in which each of the lifetime parameters expires, are considered. Instead of using lifetimes in the reference type, like Rust's type system, our type system uses sets of states Σ, to represent the set of states where the reference could expire. A reference from the Rust type system could be interpreted as a reference from our type system whose state-set is the singleton set containing the expiry state of its lifetime.

Our type system introduces the $*\Sigma\, t$ type to represent raw pointers that were cast from a $\&\Sigma$ mut t. The $\&\Sigma$ mut t type keeps track of Σ, the set of states where the $\&\Sigma$ mut t could expire, so that if Σ has only one element σ, then the $*\Sigma\, t$ can be dereferenced in the state σ. We also introduce at::$\langle\sigma\rangle(e)$ expressions that are

$$x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad l \in \textsc{LifeTime}, \quad \sigma \in \textsc{State}, \quad \Sigma \in \mathscr{P}(\textsc{State}),$$
$$t \in \textsc{Type}, \quad e \in \textsc{Exp}, \quad \Gamma \in \textsc{Var} \to (\textsc{Type} \times \textsc{State})$$
$$sc \in \textsc{SpecificationConstruct}, \quad sig \in \textsc{Signature},$$
$$spec \in \textsc{Specification}$$

$$\begin{aligned}
\sigma &::= \text{end_of}\langle l \rangle \mid \text{'pre} \mid \text{'post} \\
t &::= \&\Sigma\, t \mid \&\Sigma\, \text{mut}\, t \mid *\Sigma\, t \mid \text{Box}\langle t \rangle \mid \text{u32} \mid (t,t) \mid () \mid \text{Either}\langle t,t \rangle \mid ! \\
e &::= x \mid \text{let}\, x = e;\, e \mid *e \mid \&e \mid \& * e \mid n \mid \&e.0 \mid \&e.1 \mid (e,e) \mid () \\
&\quad \mid \text{match}\, e\, \{\text{Left}(x) => e,\ \text{Right}(x) => e\} \\
&\quad \mid \text{Left}(e) \mid \text{Right}(e) \mid e = e \mid \text{at::}\langle \sigma \rangle(e) \\
sig &::= (\overrightarrow{x_i : t_i}) \to t_{ret} \\
sc &::= \text{requires} \mid \text{ensures} \mid \text{after_expiry}\langle l \rangle \\
spec &::= \#[sc(e)] \\
\Gamma &::= \emptyset \mid \Gamma[x := t @ \sigma]
\end{aligned}$$

Figure 2.13: Type system syntax

interpreted as if the expression e were evaluated in the state σ, and treat old(e) as sugar for at::$\langle$'pre$\rangle(e)$. The specifications themselves are built by pairing a Prusti expression with a *specification construct* that indicates which type of specification it is (precondition, postcondition, or pledge).

In addition to the changes to types, there are also subtle differences in the mathematical expression syntax related to referencing and dereferencing. For example, given that `x` has type `&(u32, u32)`, the Rust expression `x.0` would correspond to the expression $*(\&x.0)$ in our type system. If instead `x` had type `(u32, u32)`, the Rust expression `x.0` would correspond to the expression $*(\&(\&x).0)$ in our type system. Despite these differences, an actual implementation of our type system could use standard Rust syntax.

$\boxed{\sigma \to \sigma}$ $\boxed{\sigma \equiv \sigma}$ **Happens At Or Before**

$$\frac{}{\text{'pre} \to \sigma}\ (\textit{pre-before}) \qquad \frac{\sigma \neq \text{'pre}}{\text{'post} \to \sigma}\ (\textit{post-before})$$

$$\frac{l_2 \text{ outlives } l_1}{\text{end\_of}\langle l_1 \rangle \to \text{end\_of}\langle l_2 \rangle}\ (\textit{outlive-before})$$

$$\frac{\sigma \neq \text{'pre} \quad \neg\text{blocked } l_1}{\text{end\_of}\langle l_1 \rangle \to \sigma}\ (\textit{non-blocked-before}) \qquad \frac{\sigma_1 \to \sigma_2 \quad \sigma_2 \to \sigma_1}{\sigma_1 \equiv \sigma_2}\ (\textit{state-eqv})$$

Figure 2.14: Rules for happens at or before relation. The conclusion of (*non-blocked-before*) implies $\text{end\_of}\langle l_1 \rangle \equiv$ 'post because of (*post-before*)

2.3.2 Happens At Or Before

The temporal order of states is an important property in our type system. For example, two states that happen at the same time can be considered to be equivalent. In this section, we present the rules to define this ordering.

Definition 2.3.1 (Happens At Or Before)**.** Judgments of the form $\sigma_1 \to \sigma_2$ state that the state σ_1 is from a point in time that is known to be at or before the point in time that the state σ_2 comes from. The full rules are given in Figure 2.14.

Property 2.3.1. The happens-at-or-before relationship is reflexive and transitive

Definition 2.3.2 (State Equivalence)**.** Judgments of the form $\sigma_1 \equiv \sigma_2$ state that states σ_1 and σ_2 come from the same point in time, see (*state-eqv*) in Figure 2.14.

All states in our type system are relative to a single function call. Since lifetime parameters in the signature of functions denote lifetime which cannot expire until after the function ends, the poststate happens at or before the expiry of any lifetime parameter. In order to include the reflexive case, (*post-before*) is defined for all states other than the prestate. Since the prestate happens before the poststate and the poststate happens at or before all the other states, (*pre-before*) states that the prestate is the earliest state.

```
fn take_first_mut<'a, 'e, 'b: 'a>
  (self: &'e mut &'a mut [T]) -> Option<&'a mut T>;
```

Figure 2.15: A function signature containing lifetime constraints

The order in which the lifetime parameters of a function expire is determined by constraints generated by Rust's type checker, based on the signature of the function. These constraints include explicit lifetime-outlives-lifetime (`'a: 'b`) and type-outlives-lifetime (`T: 'a`) constraints written by the user. In addition, there are implicit well-formedness constraints (`WF(T)`) inferred from the argument and return types. In Rust, these constraints are expanded by the compiler to create additional lifetime-outlives-lifetime constraints. At a high level, constraints of the form `WF(T)` expand to the set of constraints, `U: 'a` for each `U` and `'a` where `T` contains the reference `&'a U` or `&'a mut U`. Constraints of the form `T: 'a` expand to the set of constraints, `'b: 'a` for each `'b` that appears in `T`.

For example, in Figure 2.15 there is one user-specified constraint: `'b: 'a`, as well as the well-formedness constraint of the argument type:
`WF(&'e mut &'a mut [T])`, and of the return type:
`WF(Option<&'a mut T>)`. Following Rust's rules described above, these constraints expand to create the following set of lifetime-outlives-lifetime constraints: {`'b:'a`, `'a:'e`}.

Definition 2.3.3 (Lifetime Constraint Set)**.** The lifetime constraint set written L is the expanded set of constraints for the function being specified, which was generated by Rust's type system.

Definition 2.3.4 (Lifetime Outlives)**.** The lifetime outlives relation, which is written as l_1 outlives l_2, describes that the lifetime-outlives-lifetime constraint $l_1 \colon l_2$ is contained in the reflexive transitive closure of L.

If l_1 outlives l_2, Rust does not allow l_1 to expire until after l_2 expires, so in our type system, the rule (*outlive-before*) (Figure 2.14) states that l_2's expiry state happens at or before l_1's.

Certain lifetime parameters of a function are always free to expire as soon as the function returns, so it would be useful to consider the expiry state of these life-

times as equivalent to the poststate. Conversely, a lifetime parameter is blocked if it can be constrained to outlive another lifetime from the context where the function was called. We formalize this notion by considering the subtyping constraints (`T <: U`) that would be generated by Rust's type system if the function was called in an arbitrary valid context. Expanding these subtyping constraints creates a set of lifetime-outlives-lifetime constraints that relate the lifetime parameters of the functions with the lifetime variables from this context.

Definition 2.3.5 (Lifetime Subtyping Constraint Set)**.** The lifetime subtyping constraint set, written *LS*, is the expanded set of constraints that are generated by Rust's type system when calling the function being verified in an arbitrary context.

The subtyping constraints generated for Figure 2.15 are:
`&'x0 mut &'x1 mut [T] <: &'e mut &'a mut [T]` and
`Option<&'a mut T> <: Option<&'x2 mut T>`. These constraints expand to create the following set of lifetime-outlives-lifetime constraints:
$LS = \{$`'x0:'e`, `'x1:'a`, `'a:'x1`, `'a:'x2`$\}$.

Definition 2.3.6 (Directly Blocked)**.** A lifetime parameter l is directly blocked if *LS* contains $l : l_x$ for some lifetime variable l_x.

For the function from Figure 2.15: `'a` would be directly blocked, since *LS* contains `'a: 'x1`. Neither `'b` nor `'e` would be directly blocked since *LS* does not contain any constraints of the form `'b: _` or `'e: _`.

Definition 2.3.7 (Blocked)**.** A lifetime parameter l is blocked, written blocked l, if it outlives a lifetime parameter l_2 (as in Definition 2.3.4), such that l_2 is directly blocked.

For the function from Figure 2.15: `'a` would be blocked since `'a` outlives itself and is directly blocked. `'b` would be blocked since `'b` outlives `'a` which is directly blocked. `'e` would not be blocked since `'e` only outlives itself, and it is not directly blocked.

Since a lifetime parameter that is not blocked is only ever constrained to outlive other non-blocked lifetime parameters, it would not affect the satisfiability of the constraints to have all non-blocked lifetimes expire immediately after the call.

The rule (*non-blocked-before*), states that the expiry states of non-blocked lifetime parameters happen before all states other than the prestate.

2.3.3 Value has Type in State

It is common for programming languages to have a notion of values having types, but for our type system, this notion is dependent on the current state, i.e., a value can have a type in some states but not others. Our type system's notion of types is also slightly different than Rust's notion of types in that it only focuses on their dereferencing capabilities. For the rest of this chapter, we use this notion of types when we refer to the term "type". In the following example:

```
fn box_into_inner<T>(b: Box<T>) -> T {
  Box::into_inner(b)
}
```

`b` has type `Box<T>` in the prestate but does not in the poststate since it can no longer be dereferenced (i.e., because the memory it is pointing to has been freed at that point).

An important property in our type system is whether a value can be given a particular type in a given state, written $(v : t @ \sigma)$.[4] A value can have multiple types in the same state, for example, if a value has type t in a state it also has all of t's supertypes in the same state.

The value has a type in a state property is conceptual; the property is not checked in the type system, but the idea of the property is useful for understanding other definitions later in this chapter. Some examples of this definition for specific types are:

- A value v has the type $\text{Box}\langle t\rangle$ in the state σ if dereferencing it in σ is valid and produces a value v' such that v' has type t in σ.
  $(v : \text{Box}\langle t\rangle @ \sigma) = \exists v'.\sigma[v] = v' \wedge (v' : t @ \sigma)$

[4]This definition is related to RustBelt's [9] ownership predicate. $(v : t @ \sigma) = \lfloor t.\text{own}(v)\rfloor(\sigma)$

[5][6]where $\lfloor l \mapsto v\rfloor(\sigma) = (\sigma[l] = v)$, $\lfloor \&\text{full}\{\kappa\}P\rfloor(\sigma) = \lfloor P\rfloor(\sigma) \wedge \lfloor P\rfloor(\text{end\_of}\langle\kappa\rangle)$, $\lfloor \&\text{frac}\{\kappa\}P\rfloor(\sigma) = \forall \sigma, \sigma \to \sigma' \implies \sigma' \to \text{end\_of}\langle\kappa\rangle \implies \lfloor P\rfloor(\sigma')$

[5]This would also require the sharing predicate for mutable references to imply that they are accessible in their expiry state. If this definition were to be extended to types with interior mutability this definition would instead need to be $(v : t @ \sigma) = \lfloor t.\text{own}(v)\rfloor(\sigma) \vee \exists \kappa, l.\sigma \to \text{end\_of}\langle\kappa\rangle \wedge \sigma[l] = v \wedge \lfloor t.\text{shr}(\kappa, l)\rfloor(\sigma)$

- A value v has the type $\&\sigma_l$ mut t in the state σ if dereferencing v in σ is valid and produces a value v' such that v' has type t in σ and dereferencing v in σ_l is also valid and produces a (possibly different) value v'' such that v'' has type t in σ_l. A value v has the type $\&\Sigma_l$ mut t in the state σ if it has the type $\&\sigma_l$ mut t in the state σ for some $\sigma_l \in \Sigma_l$.
  $(v : \&\Sigma_l \text{ mut } t @ \sigma) = \exists \sigma_l \in \Sigma_l.(\exists v'.\sigma[v] = v' \wedge (v' : t @ \sigma)) \wedge (\exists v'.\sigma_l[v] = v' = v' \wedge (v' : t @ \sigma_l))$

- A value v has the type $*\sigma_l\ t$ in the state σ if dereferencing v in σ_l is valid and produces a value v' such that v' has type t in σ_l. A value v has the type $*\Sigma_l\ t$ in the state σ if it has the type $*\sigma_l\ t$ in the state σ for some $\sigma_l \in \Sigma_l$.
  $(v : *\Sigma_l\ t @ \sigma) = \exists \sigma_l \in \Sigma_l.\exists v'.\sigma_l[v] = v' \wedge (v' : t @ \sigma_l)$

- A value v has the type $\&\sigma_l$ Box$\langle$u32$\rangle$ if for all states σ' that happen between σ and σ_l dereferencing it twice in σ is valid and always produces the same value v'. A value v has the type $\&\Sigma_l$ Box$\langle$u32$\rangle$ in the state σ if it has type $\&\sigma_l$ Box$\langle$u32$\rangle$ in the state σ for some $\sigma_l \in \Sigma_l$.
  $(v : \&\Sigma_l \text{ Box}\langle\text{u32}\rangle @ \sigma) = \exists \sigma_l \in \Sigma_l.\exists v', n.\forall \sigma'.\sigma \rightarrow \sigma' \implies \sigma' \rightarrow \sigma \implies \sigma'[v] = v' \wedge \sigma[v'] = n$

2.3.4 Subtyping

The subtyping rules are similar to Rust's with a few differences. Rather than using the lifetime outlives relation for subtyping references, our type system checks that the set of expiry states for the first reference is a subset of that for the second reference. Our type system does not use the lifetime outlives relation since our type system is required to track more precisely the state when a mutable reference expires, rather than just a state the mutable reference expires after. By keeping track of sets of possible states, our type system can join types that are the same up to expiry states. This capability is shown in the following example:

```
//(if b {r1} else {r2}) has type &{'a,'b} mut u32
#[requires(*(if b {r1} else {r2}) == 0)]
fn two_ref<'a, 'b>(r1: &'a mut u32, r2: &'b mut u32,
  b: bool) -> &'b mut u32;
```

$\boxed{\Sigma <: \Sigma}$ $\boxed{t <: t}$ **Subtyping**

$$\frac{\forall \sigma_1 \in \Sigma_1 . \exists \sigma_2 \in \Sigma_2 . \sigma_1 \equiv \sigma_2}{\Sigma_1 <: \Sigma_2} \qquad \frac{\Sigma_1 <: \Sigma_2 \quad t_1 <: t_2}{\&\Sigma_1\ t_1 <: \&\Sigma_2\ t_2}$$

$$\frac{\Sigma_1 <: \Sigma_2 \quad t_1 <: t_2}{\&\Sigma_1\ \text{mut}\ t_1 <: \&\Sigma_2\ \text{mut}\ t_2} \qquad \frac{\Sigma_1 <: \Sigma_2 \quad t_1 <: t_2}{*\Sigma_1\ t_1 <: *\Sigma_2\ t_2} \qquad \frac{t_1 <: t_2}{\text{Box}\langle t_1 \rangle <: \text{Box}\langle t_2 \rangle}$$

$$\frac{}{\text{u32} <: \text{u32}} \qquad \frac{t_{11} <: t_{12} \quad t_{21} <: t_{22}}{(t_{11}, t_{21}) <: (t_{12}, t_{22})} \qquad \frac{}{() <: ()}$$

$$\frac{t_{11} <: t_{12} \quad t_{21} <: t_{22}}{\text{Either}\langle t_{11}, t_{21} \rangle <: \text{Either}\langle t_{12}, t_{22} \rangle} \qquad \frac{}{! <: !}$$

Figure 2.16: Rules for subtyping

where the specification is able to type check, even though the arms of the `if` expression have different types.

Another difference is that mutable references are covariant over their type parameter, unlike in Rust's type system where they are invariant. This is because the specifications themselves cannot mutate through references, and the mutable reference type in this context just means that they could be mutated in the function being specified.

Definition 2.3.8 (Subtyping)**.** Judgments of the form $t_1 <: t_2$ state that the type t_1 is a subtype of t_2, that is, any value that has type t_1 in a state, also has type t_2 in the same state. The full rules are given in Figure 2.16.

The intent of this definition can be summarized by the following property.

Property 2.3.2. $t_1 <: t_2$ and $(v : t @ \sigma)$ implies $(v : t_2 @ \sigma)$

This and other similar properties have not been proven, since that would require an exhaustive definition of the value has a type in a state relation. The properties express the intent of the definition, and the associated rules are chosen in such a way as to follow the properties.

2.3.5 Moving Types Between States

As previously discussed, at::$\langle\sigma\rangle(e)$ expressions evaluate e to produce a value in one state, and then allow that value to be used in another state. It is therefore important to consider which values are allowed to be moved between which states.

Definition 2.3.9 (Copy)**.** Judgments of the form t : Copy state that the type t would implement `Copy`, similar to in Rust's type system, so it cannot be consumed. This implies that if a value has a type that implements `Copy` in a state then the value will continue to have that type in all future states. The full rules are given in Figure 2.17.

The intent of this definition can be summarized by the following property.

Property 2.3.3. t : Copy and $\sigma_1 \to \sigma_2$ and $(v : t@\sigma_1)$ implies $(v : t@\sigma_2)$

The current definition for a value having a type in a state means that any pointer can have the type of a shared referenced with an expired lifetime. This is somewhat necessary since by Property 2.3.3 if a pointer (value) has type $\&\Sigma\, t$ in one state it should still have that type in all future states even those after Σ expire, and in those states, we know nothing about where it points. Since we would still like values with shared reference types to be dereferenceable we include a type outlives relation to restrict which type state pairs it is meaningful to know a value has.

Definition 2.3.10 (Type Outlives)**.** Judgments of the form $t : \sigma$ state that the type t is still meaningful in the state σ, and has a similar role to the type outlives relation in Rust's type system. The full rules are given in Figure 2.17.

Property 2.3.4. $t : \sigma_1$ and $\sigma_2 \to \sigma_1$ implies $t : \sigma_2$

Definition 2.3.11 (Value Meaningfully has Type in State)**.** A value v meaningfully has the type t in state σ when it has type t in σ and $t : \sigma$ (as in Definition 2.3.10).

Certain types like u32 have the property that they do not depend at all on the state, which allows these types to implement Copy. This property also allows these types to be used in past states instead of just future states. We consider types with this property to implement a new trait Plain, where Plain is a marker trait like Copy. Shared references are an example of a type that implements Copy but not Plain, since if a shared reference was moved to an earlier state, then dereferencing the reference in that earlier state might not be possible.

$\boxed{t : \text{Copy}}$ **Copyability**

$$\frac{}{\&\Sigma\, t : \text{Copy}} \qquad \frac{}{*\Sigma\, t : \text{Copy}} \qquad \frac{}{\text{u32} : \text{Copy}} \qquad \frac{t_1 : \text{Copy} \quad t_2 : \text{Copy}}{(t_1, t_2) : \text{Copy}}$$

$$\frac{}{() : \text{Copy}} \qquad \frac{t_1 : \text{Copy} \quad t_2 : \text{Copy}}{\text{Either}\langle t_1, t_2\rangle : \text{Copy}} \qquad \frac{}{! : \text{Copy}}$$

$\boxed{\Sigma : \sigma}$ $\boxed{t : \sigma}$ **Type Outlives**

$$\frac{\forall \sigma_1 \in \Sigma_1, \sigma \to \Sigma_1}{\Sigma_1 : \sigma} \qquad \frac{\Sigma : \sigma \quad t : \sigma}{\&\Sigma\, t : \sigma} \qquad \frac{\Sigma : \sigma \quad t : \sigma}{\&\Sigma\, \text{mut}\, t : \sigma} \qquad \frac{\Sigma : \sigma \quad t : \sigma}{*\Sigma\, t : \sigma}$$

$$\frac{t : \sigma}{\text{Box}\langle t\rangle : \sigma} \qquad \frac{}{\text{u32} : \sigma} \qquad \frac{t_1 : \sigma \quad t_2 : \sigma}{(t_1, t_2) : \sigma} \qquad \frac{}{() : \sigma} \qquad \frac{t_1 : \sigma \quad t_2 : \sigma}{\text{Either}\langle t_1, t_2\rangle : \sigma} \qquad \frac{}{! : \sigma}$$

$\boxed{t : \text{Plain}}$ **Plainness**

$$\frac{}{\text{u32} : \text{Plain}} \qquad \frac{t_1 : \text{Plain} \quad t_2 : \text{Plain}}{(t_1, t_2) : \text{Plain}} \qquad \frac{}{() : \text{Plain}} \qquad \frac{t_1 : \text{Plain} \quad t_2 : \text{Plain}}{\text{Either}\langle t_1, t_2\rangle : \text{Plain}}$$

$$\frac{}{! : \text{Plain}}$$

$\boxed{t : \sigma \rightsquigarrow \sigma}$ **Can Move Between**

$$\frac{\sigma_1 \equiv \sigma_2}{t : \sigma_1 \rightsquigarrow \sigma_2}\ (\textit{move-same}) \qquad \frac{t : \text{Copy} \quad t : \sigma_2 \quad \sigma_1 \to \sigma_2}{t : \sigma_1 \rightsquigarrow \sigma_2}\ (\textit{move-copy})$$

$$\frac{t : \text{Plain}}{t : \sigma_1 \rightsquigarrow \sigma_2}\ (\textit{move-plain})$$

Figure 2.17: Rules for moving types between states

Definition 2.3.12 (Plain)**.** Judgments of the form t : Plain state that if a value has the type t in a state then it also has type t in all states. Plain can be thought of as a marker trait like Copy. The full rules are given in Figure 2.17.

The intent of this definition can be summarized by the following property.

Property 2.3.5. t : Plain and $(v : t@\sigma_1)$ implies $(v : t@\sigma_2)$

Note that types that implement Plain also implement Copy and outlive all states as summarized in the following property.

Property 2.3.6. t : Plain implies t : Copy and $t : \sigma$

Having defined the rules for copyability, type outlives, and plainness, we can now define which values can be moved between which states.

Definition 2.3.13 (Can Move Between)**.** Judgments of the form $t : \sigma_1 \rightsquigarrow \sigma_2$ state that knowing that a value meaningfully has type t in state σ_1 implies it still meaningfully has type t in state σ_2. The full rules are given in Figure 2.17.

The intent of this definition can be summarized by the following property.

Property 2.3.7. $t : \sigma_1 \rightsquigarrow \sigma_2$ and $t : \sigma_1$ and $(v : t@\sigma_1)$ implies $t : \sigma_2$ and $(v : t@\sigma_2)$

2.3.6 Typing

The definitions discussed so far, are now used to help build the core type system for typing Prusti expressions.

Definition 2.3.14 (Environment Satisfies Type Environments)**.** Let an environment Δ be a map from variables to values. We say Δ satisfies Γ according to the following rules: All environments satisfy $\emptyset$. An environment satisfies $\Gamma[x := t@\sigma]$ if it has x bound to a value that meaningfully has type t in state σ, and it also satisfies Γ.

Definition 2.3.15 (Typing)**.** Judgments of the form $\Gamma, \sigma \vdash e : t$ state that evaluating the expression e in an environment satisfying Γ in the state σ produces a value that meaningfully has type t in state σ. The full rules are given in Figure 2.18.[7]

2.3.7 Specification Well-Definedness

Definition 2.3.16 (Specification Construct Context)**.** Judgments of the form $sig, sc \vdash \Gamma, \sigma$ state that the specifications of the form sc, that are applied to functions with the signature sig, are evaluated in an environment satisfying Γ in the state σ. The full rules are given in Figure 2.19.

[7] The rule (*mut-cast*) would make our type system difficult to implement in a type checker since choosing when to apply the rule is not clear. This issue is addressed in Section 2.4.4

$\boxed{\Gamma, \sigma \vdash e : t}$ **Typing**

$$\frac{\Gamma, \sigma \vdash e : t_1 \quad t_1 <: t_2}{\Gamma, \sigma \vdash e : t_2}\ (sub) \qquad \frac{\Gamma, \sigma \vdash e_1 : t_1 \quad \Gamma[x := t_1 @ \sigma], \sigma \vdash e_2 : t_2}{\Gamma, \sigma \vdash \text{let}\, x = e_1; e_2 : t_2}\ (let)$$

$$\frac{t : \sigma_1 \rightsquigarrow \sigma_2}{\Gamma[x := t @ \sigma_1], \sigma_2 \vdash x : t}\ (var) \qquad \frac{\Gamma, \sigma \vdash e : t}{\Gamma, \sigma \vdash \&e : \&\sigma\, t}\ (shr)$$

$$\frac{\Gamma, \sigma \vdash e : \&\Sigma\, t}{\Gamma, \sigma \vdash *e : t}\ (shr\text{-}deref) \qquad \frac{\Gamma, \sigma \vdash e : \&\Sigma_1\, \&\Sigma_2\, \text{mut}\, t}{\Gamma, \sigma \vdash \& * e : \&\Sigma_1\, t}\ (mut\text{-}cur\text{-}deref)$$

$$\frac{\Gamma, \sigma \vdash e : \&\Sigma_2\, \text{mut}\, t}{\Gamma, \sigma \vdash e : *\Sigma_2\, t}\ (mut\text{-}cast) \qquad \frac{\Gamma, \sigma \vdash e : \&\Sigma_1\ *\sigma\, t}{\Gamma, \sigma \vdash \& * e : \&\Sigma_1\, t}\ (mut\text{-}fin\text{-}deref)$$

$$\frac{\Gamma, \sigma \vdash e : \&\Sigma_1\, \text{Box}\langle t \rangle}{\Gamma, \sigma \vdash \& * e : \&\Sigma_1\, t} \qquad \frac{}{\Gamma, \sigma \vdash n : \text{u32}} \qquad \frac{\Gamma, \sigma \vdash e : \&\Sigma_1\, (t_0, t_1)}{\Gamma, \sigma \vdash \&e.0 : \&\Sigma_1\, t_0}$$

$$\frac{\Gamma, \sigma \vdash e : \&\Sigma_1\, (t_0, t_1)}{\Gamma, \sigma \vdash \&e.1 : \&\Sigma_1\, t_1} \qquad \frac{\Gamma, \sigma \vdash e_1 : t_1 \quad \Gamma, \sigma \vdash e_2 : t_2}{\Gamma, \sigma \vdash (e_1, e_2) : (t_1, t_2)} \qquad \frac{}{\Gamma, \sigma \vdash () : ()}$$

$$\frac{\Gamma, \sigma \vdash e : !}{\Gamma, \sigma \vdash e : t} \qquad \frac{\begin{array}{c}\Gamma, \sigma \vdash e_0 : \&\Sigma\, \text{Either}\langle t_1, t_2 \rangle \\ \Gamma[x_1 := \&\Sigma\, t_1 @ \sigma], \sigma \vdash e_1 : t_3 \quad \Gamma[x_2 := \&\Sigma\, t_2 @ \sigma], \sigma \vdash e_2 : t_3\end{array}}{\Gamma, \sigma \vdash \text{match}\, e_0\, \{\text{Left}(x_1) => e_1,\ \text{Right}(x_2) => e_2\} : t_3}$$

$$\frac{\Gamma, \sigma \vdash e : t_1}{\Gamma, \sigma \vdash \text{Left}(e) : \text{Either}\langle t_1, t_2 \rangle} \qquad \frac{\Gamma, \sigma \vdash e : t_2}{\Gamma, \sigma \vdash \text{Right}(e) : \text{Either}\langle t_1, t_2 \rangle}$$

$$\frac{\Gamma, \sigma_1 \vdash e : t \quad t : \sigma_1 \rightsquigarrow \sigma_2}{\Gamma, \sigma_2 \vdash \text{at::}\langle \sigma_1 \rangle (e) : t}\ (state\text{-}change)$$

Figure 2.18: Rules for typing Prusti expression

Definition 2.3.17 (Specification Well-Definedness)**.** Judgments of the form $sig \vdash spec$ state that the specification *spec* is well-defined for functions with the signature *sig*. The rule is given in Figure 2.19.

The final type system rules from Figure 2.19 define well-definedness for Prusti

$\boxed{sig, sc \vdash \Gamma, \sigma}$ **Specification Construct Context**

$$\frac{\Gamma = \emptyset[\overrightarrow{x_i := t_i @\text{'pre}}]}{(\overrightarrow{x_i : t_i,}) \rightarrow t_{ret}, \text{requires} \vdash \Gamma, \text{'pre}} \ (\textit{precondition})$$

$$\frac{\Gamma = \emptyset[\overrightarrow{x_i := t_i @\text{'pre}}][\text{result} := t_{ret} @\text{'post}]}{(\overrightarrow{x_i : t_i,}) \rightarrow t_{ret}, \text{ensures} \vdash \Gamma, \text{'post}} \ (\textit{postcondition})$$

$$\frac{\Gamma = \emptyset[\overrightarrow{x_i := t_i @\text{'pre}}][\text{result} := t_{ret} @\text{'post}]}{(\overrightarrow{x_i : t_i,}) \rightarrow t_{ret}, \text{after\_expiry}\langle l \rangle \vdash \Gamma, \text{end\_of}\langle l \rangle} \ (\textit{pledge})$$

$\boxed{sig \vdash spec}$ **Specification Well-Definedness**

$$\frac{sig, sc \vdash \Gamma, \sigma \quad \Gamma, \sigma \vdash e : \text{bool}}{sig \vdash \#[sc(e)]}$$

Figure 2.19: Rules for specification well-definedness

specifications. The main rule states that a specification is well-defined if its body has the boolean type in a particular type environment Γ and state σ, determined by the type of specification and by the signature of the function being specified. For the rule (*precondition*) σ is the prestate, for (*postcondition*) σ is the poststate, and for (*pledge*) σ is the expiry state of the specified lifetime. The initial environment for evaluating specifications has the arguments bound in the prestate and if that specification is a postcondition or pledge it also has the return value (`result`) bound in the poststate. We can now use our type system to give us the set of well-defined Prusti specifications.

2.4 Type System Extensions

In this section, we extend our type system, introduced in the previous section, in several ways to expand the set of specifications that are considered well-defined. These extensions allow more types of behaviours to be specified and make writing specifications more convenient.

```
#[predicate]
fn unchanged<'s>(x: u32) -> bool {
  at::<'s>(x) == x
}
```

```
#[ensures(unchanged::<'pre>(*m)]
fn test<'a>(m: &'a mut u32)
```

Call by value inline:

```
#[ensures({
  let x = *m;
  at::<'pre>(x) == x
})]
fn test<'a>(m: &'a mut u32)
```

Call by name inline:

```
#[ensures(at::<'pre>(*m) == *m)]
fn test<'a>(m: &'a mut u32)
```

Figure 2.20: Comparison of function call semantics

2.4.1 Functions

It is possible to extend our type system in various ways to support function types in specifications. An important distinction when deciding how to do this is whether specification-level functions have "call-by-name" like, or "call-by-value" like, semantics. This affects how lambda-bound variables interact with `at` expressions as seen in Figure 2.20. In the call-by-value inlined example `at::<'pre>(x)` is equivalent to `x` since the state in which variables are evaluated does not matter (the evaluation state only affects dereferencing), which causes the call-by-value version to be equivalent to `true`. The call-by-name version asserts that the value under the mutable reference was left unchanged.

Since Rust has call-by-value semantics, we choose the call-by-value semantics for the specification language as well. The rules for functions are shown in Figure 2.21. Our type system requires that the argument of a function is checked in the current state since it is eagerly evaluated. Lambda expressions are checked as if they were run in the state they are created in. Function types do not implement Copy so functions cannot be moved to another state which ensures that they will

Function Simple

$$\frac{\sigma_1 \equiv \sigma_2 \quad t_{12} <: t_{11} \quad t_{21} <: t_{22}}{\mathrm{Fn}(t_{11} @ \sigma_1) \to t_{21} <: \mathrm{Fn}(t_{12} @ \sigma_2) \to t_{22}} \; (\textit{fun-sub})$$

$$\frac{\Gamma[x := t_1 @ \sigma_2], \sigma_1 \vdash e : t_2}{\Gamma, \sigma_1 \vdash |x\colon t_1 @ \sigma_2|\; e : \mathrm{Fn}(t_1 @ \sigma_2) \to t_2} \; (\textit{lam})$$

$$\frac{\Gamma, \sigma \vdash e_1 : \mathrm{Fn}(t_1 @ \sigma) \to t_2 \quad \Gamma, \sigma \vdash e_2 : t_1}{\Gamma, \sigma \vdash e_1(e_2) : t_2} \; (\textit{app})$$

Figure 2.21: Initial rules for functions

```
#[logic]
fn increases<'s>(x: &'curr mut u32@'s) -> bool {
  at::<'s>(*x) <= *x
}
```

```
#[ensures(increases::<'pre>(old(p.0)))]
fn test<'a>(p: (&'a mut u32, u32))
```

Call by value inline:

```
#[ensures({
  let x = old(p.0);
  at::<'pre>(*x) < *x
})]
fn test<'a>(p: (&'a mut u32, u32))
```

Figure 2.22: Specification that calls a function with an argument from another state

always be run in the state where they were created.

2.4.2 Function Arguments from Different States

In some cases, it can be useful to extract a relation involving multiple states into a helper function that can be called in specifications, for example, `increases` from Figure 2.22. The `increases` function is a specification-only function that takes in `'s`, a state, and `x`, an `&'curr mut u32@'s` which is a mutable reference

$\boxed{\Gamma, \sigma_1 \vdash e : t@\sigma_2}$ **Function Arguments from Different States**

$$\Gamma, \sigma \vdash e : t := \Gamma, \sigma \vdash e : t@\sigma$$

$$\frac{\Gamma, \sigma_2 \vdash e_1 : t_1@\sigma_1 \quad \Gamma[x := t_1@\sigma_1], \sigma_2 \vdash e_2 : t_2}{\Gamma, \sigma \vdash \text{let}\, x = e_1; e_2 : t_2} \; (\textit{let'})$$

$$\frac{\Gamma, \sigma_1 \vdash e_1 : \text{Fn}(t_1@\sigma_2) \to t_2 \quad \Gamma, \sigma_1 \vdash e_2 : t_1@\sigma_2}{\Gamma, \sigma_1 \vdash e_1(e_2) : t_2} \; (\textit{app'})$$

$$\frac{\Gamma, \sigma_1 \vdash e : t \quad t : \sigma_1 \rightsquigarrow \sigma_2}{\Gamma, \sigma_3 \vdash \text{at}::\langle\sigma_1\rangle(e) : t@\sigma_2} \; (\textit{state-change'})$$

Figure 2.23: Rules for calling functions with arguments from different states. The original versions of (*var*) and (*state-change*) can be obtained by instantiating the new versions with $\sigma_2 = \sigma_3$. The new versions are more useful since they can be combined with (*let'*) or (*app'*).

that was passed in from the state `'s` and expires in the current state. In our type system this would be expressed as:

$$\Gamma, \sigma_{curr} \vdash \text{increases}::\langle\sigma_s\rangle : \text{Fn}(\&\sigma_{curr}\ \text{mut u32}@\sigma_s) \to \text{bool}$$

The function asserts that `x` has a larger value now than it did in state `'s`. The specification for the `test` function calls this helper function using the prestate and field 0 of its parameter.

In the type system with the simple function extension shown in Figure 2.21, the application would not type check. `p.0` would not be able to escape the `old` expression without converting to an ∗'a u32, and then `old(p.0)` would not be able to move back from the poststate to the prestate in order to be used by the call. These two problems are in some sense opposites. If the argument did not need to pass through the poststate and could go straight from the prestate in the `old` expression to the prestate required by the function, then this example would be able to type check.

By extending our type system to allow arguments to slip from one state to

```
#[ensures(match result.last() {
  Some(x) => x == old(e), // error
  None => false
}]
fn vec_push<T>(v: Vec<T>, e: T) -> Vec<T> {
  v.push(e);
  v
}
```

Figure 2.24: Function with non-`Copy` parameters

another while avoiding the current state in a few specific cases, it is possible to allow this type of function and maintain call-by-value semantics. This is shown in Figure 2.23 which uses Definition 2.4.1.

Definition 2.4.1 (Generalized Typing)**.** Judgments of the form $\Gamma, \sigma_1 \vdash e : t@\sigma_2$ state that evaluating the expression e in an environment satisfying Γ (see Definition 2.3.14) in the state σ_1 produces a value that meaningfully has type t in state σ_2. Judgments of the form $\Gamma, \sigma \vdash e : t$ can be viewed as sugar for $\Gamma, \sigma \vdash e : t@\sigma$. .

The rules (*app'*) and (*state-change'*) are the main rules that are needed for this. The (*let'*) rule mirrors this change so that inlining functions into let-expressions preservers typing.

2.4.3 Equality

We define equality in our type system, such that the equality operator can accept arguments from different states, similar to the functions described in Section 2.4.2. Without defining equality this way, if a function's parameter's types contain type parameters that do not implement `Copy`, writing postconditions that relate the result of the function to the function's parameters would not be possible. Writing these postconditions would not be possible since the parameters and return value come from different states but would need to be moved to the same state in order to be compared (types that do not implement `Copy` cannot be moved to different states as shown in Figure 2.17). For example, the specification shown in Figure 2.24 attempts to compare the parameter `e` with the last element of the returned

vector `result`. This comparison cannot be done in the prestate since `result` has not been created at that point, but it also cannot be done in the poststate since `e` would have already been consumed at that point. By allowing the equality operator to accept arguments from different states, we avoid this issue.

The implementation of this equality operator can be thought of as taking a *snapshot* (as described in [Wolff et al. 15]) of each of its parameters in their respective states, and then checking if the snapshots are identical.

For pointer types, there are two reasonable ways to implement snapshotting. The first is to only include the snapshot of the dereference of the pointer. The second one is to also include the pointer itself.

For mutable references, we decide to include the pointer. We want mutable references only to be considered equal if they share the same pointer, so that dereferencing two equal mutable references, in the same state, will give the same result.

In the following example:

```
#[ensures(result == old(x) ==> *result == old(*x)]
#[ensures(result == old(x) ==> at::<'a>(*result == *x)]
fn test<'a, T>(x: &'a mut T) -> &'a mut T;
```

Knowing `result == old(x)` would imply `*result == old(*x)` since the snapshot of a mutable reference includes the snapshot of dereferencing it. It would also imply that `result` and `x` are the same pointer, which would imply that dereferencing either of them in the state `'a` would give the same result. This makes both postconditions tautologies (they will always be true).

For boxes, we decide not to include the pointer since Rust's aliasing rules[8] make it unclear when we can know that two boxes are the same pointer, so we want box equality to only depend on the dereference and not on the pointer.

For shared references, we also decide not to include the pointer, since it is unclear what this pointer should be when creating a shared reference inside a specification, since these shared references do not exist at run time. This does not apply to mutable references since we do not allow them to be created inside specifications.

For non-pointer types, the snapshot operation is implemented by recursively snapshotting each inner value. Figure 2.25 shows the definition of the snap opera-

[8] https://github.com/rust-lang/unsafe-code-guidelines/issues/326

Pointer Types

$$\text{snap}(x : \&\Sigma\,\text{mut}\,t) := (x, *x) \qquad \text{snap}(x : \text{Box}\langle t\rangle) := *x \qquad \text{snap}(x : \&\Sigma\,t) := *x$$

Non-Pointer Types

$$\text{snap}(x : \text{u32}) := x \qquad \text{snap}(x : (t_1, t_2)) := (\text{snap}(x.0), \text{snap}(x.1))$$

$$\text{snap}(x : ()) := () \qquad \text{snap}(x : \text{Either}\langle t_1, t_2\rangle) :=$$

$$\text{match}\ x\ \{\text{Left}(l) \Rightarrow \text{Left}(\text{snap}(l)),\ \text{Right}(r) \Rightarrow \text{Right}(\text{snap}(r))\}$$

$$\text{snap}(x : !) := x$$

Figure 2.25: Snapshot implementation

tion of each "snapshottable" type in our type system.

In Section 2.3.1 we introduced the $*\Sigma\,t$ type to represent raw pointers that track their expiry state. Since values that have the type $*\Sigma\,t$ cannot necessarily be dereferenced in the current state, there is no good way to handle snapshotting this type so we do not allow snapshotting types containing the type $*\Sigma\,t$. We also do not consider our function types (introduced in Section 2.4.1) to be snapshottable, but since they represent specification functions they are not valid Rust types. Luckily since the type $*\Sigma\,t$ is not a valid Rust type either, all valid Rust types are snapshottable, so the type parameters of run-time functions are always snapshottable.

Definition 2.4.2 (snapshottability)**.** Judgments of the form t : CanSnap state that values of the type t can have snapshots taken of them, which allows them to be checked for equality. CanSnap can be thought of as a marker trait like Copy. The full rules are given in Figure 2.26.

Figure 2.26 shows the extra rules that need to be added to our type system to support this equality feature.

2.4.4 Zombie

In specifications, it is arguably reasonable to use part of a non-`Copy` type that has been consumed, as long as the part being used is `Copy`. In the following example:

$\boxed{t : \text{CanSnap}}$ **Snapshottability**

$$\frac{t : \text{CanSnap}}{\&\Sigma\, t : \text{CanSnap}} \qquad \frac{t : \text{CanSnap}}{\&\Sigma\, \text{mut}\, t : \text{CanSnap}} \qquad \frac{t : \text{CanSnap}}{\text{Box}\langle t\rangle : \text{CanSnap}}$$

$$\frac{}{\text{u32} : \text{CanSnap}} \qquad \frac{t_1 : \text{CanSnap} \quad t_2 : \text{CanSnap}}{(t_1, t_2) : \text{CanSnap}} \qquad \frac{}{() : \text{CanSnap}}$$

$$\frac{t_1 : \text{CanSnap} \quad t_2 : \text{CanSnap}}{\text{Either}\langle t_1, t_2\rangle : \text{CanSnap}} \qquad \frac{}{! : \text{CanSnap}}$$

Equality

$$\frac{\Gamma, \sigma \vdash e_1 : t@\sigma_1 \quad \Gamma, \sigma \vdash e_2 : t@\sigma_2 \quad t : \text{CanSnap}}{\Gamma, \sigma \vdash e_1 = e_2 : \text{bool}}$$

Figure 2.26: Rules for Equality

Zombie Extension

$$*\Sigma\, t := \text{Zombie}\langle \&\Sigma\, \text{mut}\, t\rangle$$

$$\frac{}{\text{Zombie}\langle t\rangle : \text{Copy}}\ (\textit{copy-zombie}) \qquad \frac{t : \sigma}{\text{Zombie}\langle t\rangle : \sigma}\ (\textit{outlive-zombie})$$

$$\frac{t_1 <: t_2}{t_1 <: \text{Zombie}\langle t_2\rangle}\ (\textit{make-zombie}) \qquad \frac{t_1 <: t_2}{\text{Zombie}\langle t_1\rangle <: \text{Zombie}\langle t_2\rangle}\ (\textit{cov-zombie})$$

$$\frac{t_2 : \sigma_1 \rightsquigarrow \sigma_2 \quad t_1 <: t_2}{\Gamma[x := t_1@\sigma_1], \sigma_2 \vdash x : t_2}\ (\textit{var-sub})$$

Figure 2.27: Rules for zombies

```
#[ensures(result == x.0)]
// vs #[ensures(result == old(x.0)]
fn(x: (u32, Box<u32>)) -> u32
```

`x` is consumed by the call since it has type `(u32, Box<u32>)` (which does not implement `Copy`), but since `x.0` is the only part of `x` being used, and its

type, `u32`, is `Copy` this specification be considered valid. These types of usages generally allow specifications to be more concise by avoiding extra 'at::$\langle\sigma\rangle(e)$' expressions like the `old(x.0)` on line 2.

In order to support these types of usages we introduce a Zombie$\langle t\rangle$ type as shown in Figure 2.27 that allows masking non-`Copy` parts of a type, making them `Copy`, but stopping their values from being useable. Since not all parts of a Zombie$\langle t\rangle$ are usable, we do not include a rule to allow zombie types to be snapshotted.

Typing the above example makes use of the `var-sub` rule for `x`:

$$\frac{t_2 : \text{'pre} \rightsquigarrow \text{'post} \quad (\text{u32}, \text{Box}\langle\text{u32}\rangle) <: t_2}{\Gamma[\text{x} := (\text{u32}, \text{Box}\langle\text{u32}\rangle)@\text{'pre}], \text{'post} \vdash \text{x} : t_2}\ (\textit{var-sub})$$

This still leaves open how to instantiate t_2. Using $(\text{u32}, \text{Box}\langle\text{u32}\rangle)$ does not work since we cannot prove $(\text{u32}, \text{Box}\langle\text{u32}\rangle) : \text{'pre} \rightsquigarrow \text{'post}$ (since it would require $(\text{u32}, \text{Box}\langle\text{u32}\rangle) : \text{Copy}$). Using Zombie$\langle(\text{u32}, \text{Box}\langle\text{u32}\rangle)\rangle$ works for this rule but causes the proof to fail later since Zombie$\langle(\text{u32}, \text{Box}\langle\text{u32}\rangle)\rangle$ does not support projection. The correct instantiation is actually $(\text{u32}, \text{Zombie}\langle\text{Box}\langle\text{u32}\rangle\rangle)$. In general, it is possible to eagerly choose this instantiation (t_2) by covering the smallest possible part of the original type (t_1) with Zombie$\langle\cdot\rangle$ while still making it Copy. This can be done by wrapping each Box$\langle t\rangle$, $\&\Sigma$ mut t, and Fn$(t@\sigma) \to t$ that is not under a Snap$\langle\cdot\rangle$, Zombie$\langle\cdot\rangle$, or $\&\Sigma\ \cdot$, in a Zombie$\langle\cdot\rangle$.

Another similar use case would be to dereference a mutable reference that is part of a larger type when it expires, even though the type would have been consumed at that point, for example:

```
#[ensures(*x.0 == x.1)]
// vs #[ensures(*old(x.0) == old(x.1)]
fn test(x: (&mut u32, u32))
```

The zombie extension can support this by making the type $*\Sigma\ t$ be sugar for Zombie$\langle\&\Sigma$ mut $t\rangle$ instead of an independent type. This would allow the rule (*mut-cast*) to be removed and make type checking easier for mutable references since they could lazily be cast into zombies via subtyping as soon as a state change

$$x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad l \in \textsc{LifeTime}, \quad \sigma \in \textsc{State}, \quad \Sigma \in \mathscr{P}(\textsc{State}),$$
$$t \in \textsc{Type}, \quad e \in \textsc{Exp}, \quad sc \in \textsc{SpecificationConstruct}$$

$$\begin{aligned}
\sigma ::= &\ \text{end_of}\langle l \rangle \mid \text{'pre} \mid \text{'post} \\
t ::= &\ \&\Sigma\, t \mid \&\Sigma\, \text{mut}\, t \mid \text{Box}\langle t \rangle \mid \text{u32} \mid (t,t) \mid () \mid \text{Either}\langle t,t \rangle \mid ! \\
&\mid \text{Fn}(t@\sigma) \to t \mid \text{Zombie}\langle t \rangle \\
e ::= &\ x \mid \text{let}\, x = e;\, e \mid *e \mid \&e \mid \& * e \mid n \mid \&e.0 \mid \&e.1 \mid (e,e) \mid () \\
&\mid \text{match}\, e\, \{\text{Left}(x) \Rightarrow e,\ \text{Right}(x) \Rightarrow e\} \mid \text{Left}(e) \mid \text{Right}(e) \mid e = e \\
&\mid \text{at::}\langle \sigma \rangle(e) \mid (|x\colon t@\sigma|\ e) \mid e(e) \\
sc ::= &\ \text{requires} \mid \text{ensures} \mid \text{after_expiry}\langle l \rangle
\end{aligned}$$

Figure 2.28: Prusti Syntax with Extensions

occurs.

2.5 Prusti to Creusot Translation

Now that we have a type system for Prusti specifications we can use it to guide the translation to Creusot specifications. On the surface Creusot specifications (see Figure 2.29) look similar to Prusti specifications (see Figure 2.28), so the translation leaves the Prusti specifications mostly unchanged and only focuses on the differences.

Conceptually, Prusti types can be translated by erasing the Zombie$\langle\cdot\rangle$ type constructor and the expiry state sets from references as shown in Figure 2.30. Because our type system only uses subtyping for expiry state sets and Zombies, which are erased when translating to Creusot, if one Prusti type is a subtype of another Prusti type, then both translate to the same Creusot type. This is summarized in the following property.

Property 2.5.1. $t_1 <: t_2$ implies $[\![t_1]\!] = [\![t_2]\!]$

The key challenge in defining the translation is deciding when to use Creusot's ^ (final) operator as opposed to its $*$ (current) operator. Instead of directly translating

$$x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad ct \in \textsc{CreusotType}, \quad ce \in \textsc{CreusotExp},$$
$$csc \in \textsc{CreusotSpecificationConstruct}$$

$$\begin{aligned}
ct ::= &\ \&\, ct \mid \&\, \text{mut}\, ct \mid \text{Box}\langle ct\rangle \mid \text{u32} \mid (ct, ct) \mid () \mid \text{Either}\langle ct, ct\rangle \mid ! \mid \text{Fn}(ct) \to ct \\
ce ::= &\ x \mid \text{let}\, x = ce;\, ce \mid *e \mid \&ce \mid \&*ce \mid \&\hat{}ce \mid n \mid \&ce.0 \mid \&ce.1 \mid (ce, ce) \mid () \\
&\mid \text{match}\, ce\, \{\text{Left}(x) \Rightarrow ce,\ \text{Right}(x) \Rightarrow ce\} \mid \text{Left}(ce) \mid \text{Right}(ce) \\
&\mid ce = ce \mid (|x\colon t|\ ce) \mid ce(ce) \\
csc ::= &\ \text{requires} \mid \text{ensures}
\end{aligned}$$

Figure 2.29: Creusot Syntax

$\boxed{[\![t]\!] = ct}$ **Type Translation**

$$[\![\&\Sigma\, t]\!] = \&\ [\![t]\!] \qquad [\![\&\Sigma\, \text{mut}\, t]\!] = \&\ \text{mut}\ [\![t]\!] \qquad [\![\text{Box}\langle t\rangle]\!] = \text{Box}\langle [\![t]\!]\rangle$$

$$[\![\text{u32}]\!] = \text{u32} \qquad [\![(t_1, t_2)]\!] = ([\![t_1]\!], [\![t_2]\!]) \qquad [\![()]\!] = ()$$

$$[\![\text{Either}\langle t_1, t_2\rangle]\!] = \text{Either}\langle [\![t_1]\!], [\![t_2]\!]\rangle \qquad [\![!]\!] = !$$

$$[\![\text{Fn}(t_1 @ \Sigma) \to t_2]\!] = \text{Fn}([\![t_1]\!]) \to [\![t_2]\!] \qquad [\![\text{Zombie}\langle t\rangle]\!] = [\![t]\!]$$

Figure 2.30: Type Translation ($[\![\cdot]\!]$ is the translation function)

Prusti specifications to Creusot specifications, we first type check the Prusti specification to create a derivation tree for the specification well-definedness judgment (from Figure 2.19), and define our translation over the derivation tree itself. This makes the translation relatively straightforward, by allowing us to choose which operator to use based on how the Prusti expression was typed. The Prusti specification construct is directly translated, by translating 'after_expiry$\langle l\rangle$' to 'ensures', and otherwise leaving it unchanged. The type derivation tree for the Prusti expression is recursively translated to produce the Creusot expression for the specification. These steps are summarized in Figure 2.31.

When translating the type derivation tree, most of the rules for type checking a

$\boxed{[\![sc]\!] = csc}$ **Specification Construct Translation**

$[\![\text{requires}]\!] = \text{requires}$ $\quad$ $[\![\text{ensures}]\!] = \text{ensures}$ $\quad$ $[\![\text{after\_expiry}\langle l\rangle]\!] = \text{ensures}$

$\boxed{[\![\mathscr{S}]\!] = cspec}$ **Specification Derivation Translation**

$$\left[\!\!\left[\frac{sig, sc \vdash \Gamma, \sigma \quad \mathscr{D} :: \Gamma, \sigma \vdash e : \text{bool}}{sig \vdash \#[sc(e)]}\right]\!\!\right] = \#[[\![sc]\!]([\![\mathscr{D}]\!])]$$

Figure 2.31: Specification Translation ($[\![\cdot]\!]$ is the translation function)

$\boxed{[\![\mathscr{D}]\!] = ce}$ **Type Derivation Translation**

$$\left[\!\!\left[\frac{\mathscr{D} :: \Gamma, \sigma_1 \vdash e : t \quad t : \sigma_1 \rightsquigarrow \sigma_2 \quad \sigma_1 \rightarrow \sigma_2}{\Gamma, \sigma_2 \vdash \text{at}::\langle\sigma_1\rangle(e) : t}\ (\textit{state-change})\right]\!\!\right] = [\![\mathscr{D}]\!]$$

$$\left[\!\!\left[\frac{\mathscr{D} :: \Gamma, \sigma \vdash e : \&\Sigma_1\ \text{Zombie}\langle\&\sigma\ \text{mut}\ t\rangle}{\Gamma, \sigma \vdash \& * e : \&\Sigma_1\ t}\ (\textit{mut-fin-deref})\right]\!\!\right] = \&\hat{}\,[\![\mathscr{D}]\!]$$

Figure 2.32: Expression Translation ($[\![\cdot]\!]$ is the translation function)

Prusti expression are translated to a Creusot expression that has the same form as the Prusti expression, but that recursively translates its inner expressions using the premises of the given rule. The following:

$$\left[\!\!\left[\frac{\mathscr{D}_1 :: \Gamma, \sigma \vdash e_1 : t_1 \quad \mathscr{D}_2 :: \Gamma, \sigma \vdash e_2 : t_2}{\Gamma, \sigma \vdash (e_1, e_2) : (t_1, t_2)}\ (\textit{pair})\right]\!\!\right] = ([\![\mathscr{D}_1]\!], [\![\mathscr{D}_2]\!])$$

shows this type of translation using the rule (*pair*) as an example. The exceptions to this pattern are the (*state-change*), and (*mut-fin-deref*) rules. The (*state-change*) rule translates to the translation of its premise. The (*mut-fin-deref*) rule translates to the $\hat{}$ operator applied to the translation of its premise. These translations are shown in Figure 2.32

In summary, at a high level, the translation works by erasing the $\text{at}::\langle\sigma\rangle(\cdot)$ operator, and by replacing Prusti's dereference operator $*$ with Creusot's $\hat{}$ (final)

```
#[requires(*x < u32::MAX)]
#[ensures(*x == old(*x) + 1)]
fn increment<'a>(x: &'a mut u32) { *x += 1; }
```

translates to

```
#[requires(*x < u32::MAX)]
#[ensures(^x == *x + 1)]
fn increment<'a>(x: &'a mut u32) { *x += 1; }
```

Figure 2.33: `increment` function specification translation

operator when the dereference was type checked by the rule (*mut-fin-deref*). Even though the at::$\langle\sigma\rangle(\cdot)$ operator is erased, it still affects the overall translation since the operator changes how the type derivation tree for its inner expression is built. More specifically the at::$\langle\sigma\rangle(\cdot)$ operator can affect whether dereference operators that it contains are typed using (*mut-curr-deref*) or (*mut-fin-deref*) as will be shown in the examples.

2.5.1 Examples

We now present a few examples of translating Prusti specifications to Creusot specifications using our translation.

Figure 2.33 shows an example of applying the translation to the specifications of the `increment` function. Type checking the preconditions does not use the (*state-change*), or (*mut-fin-deref*) rules, so the translation does not change anything. In the postcondition, the first usage of `x` requires `x` to be cast to a Zombie$\langle$&end_of$\langle$'a$\rangle$ mut u32$\rangle$ so `x` can be moved from the prestate to the poststate. This forces the dereference to be typed using the rule (*mut-fin-deref*) [9], so the dereference is translated to a (^). The old$(\cdot)$ operator, which is sugar for at::$\langle$'pre$\rangle(\cdot)$, is erased leaving its inner expression (*x) unchanged since it was typed using the rule (*mut-curr-deref*).

Figure 2.34 shows an example of applying the translation to the specifications of the `get_mut` function. The translation works similarly to the translation of the

[9] Since 'a is not blocked, 'a $\equiv$ 'post, so Zombie$\langle$&end_of$\langle$'a$\rangle$ mut u32$\rangle$ <: Zombie$\langle$&'post mut u32$\rangle$, which allows (*mut-fin-deref*) to be used

```
#[ensures(old((*x).0) == *result)]
#[after_expiry((*x).0 == *result)]
#[after_expiry((*x).1 == old((*x).1))]
fn get_mut<'a>(x: &'a mut (u32, u32)) -> &'a mut u32 {
  return &mut x.0;
}
```

translates to

```
#[ensures((*x).0 == *result)]
#[ensures((^x).0 == ^result)]
#[ensures((^x).1 == (*x).1)]
fn get_mut<'a>(x: &'a mut (u32, u32)) -> &'a mut u32 {
  return &mut x.0;
}
```

Figure 2.34: `get_mut` function specification translation

```
#[ensures(result == old(x) ==> *result == old(*x)]
#[ensures(result == old(x) ==> at::<'a>(*result == *x)]
fn test<'a, T>(x: &'a mut T) -> &'a mut T;
```

translates to

```
#[ensures(result == x ==> *result == *x)]
#[ensures(result == x ==> ^result == ^x)]
fn test<'a, T>(x: &'a mut T) -> &'a mut T;
```

Figure 2.35: Prusti Tautology that translates to a Creusot Tautology

`increment` specifications, but it also replaces the pledges (`after_expiry`) with postconditions (`ensures`).

Figure 2.35 shows an example translation that results in a tautology in Creusot (by congruence). The Prusti specification is also a tautology because of the way we defined the snapshot operation for mutable references as shown in Section 2.4.3.

2.5.2 Creusot's Mutable Reference Extensionality Axiom

Creusot has an extensionality axiom for mutable references which states that two mutable references with the same current and final values are always equal. In Prusti mutable references can only be equal if they alias each other (i.e., they are

```
#[ensures(*result == old(*x) && at::<'a>(*result == *x)
  ==> x == result]
fn test<'a, T>(x: &'a mut T) -> &'a mut T;
```

translates to

```
#[ensures(*result == *x && ^result == ^x
  ==> x == result)]
fn test<'a, T>(x: &'a mut T) -> &'a mut T;
```

Figure 2.36: Prusti specification that should not translate to a Creusot Tautology

the same pointer), which is slightly stricter than Creusot. Creusot's extensionality axiom can lead to cases where translating from Prusti to Creusot is not meaning-preserving. For example, if in Prusti, we had a specification involving two mutable references that have the same value now and will have the same value when they expire, but do not alias each other, then our equality rules (Section 2.4.3) would say that they are not equal. After translating to Creusot, however, since their current and final values are the same, according to Creusot they would be considered equal.

Figure 2.36 shows an example of this. If the value returned by the implementation of the function does not alias `x`, then the Prusti specification should not verify, according to its definition of equality, but in Creusot, the axiom causes the post-condition to be a tautology and so it would always verify.

If this extensionality axiom for mutable references was removed from Creusot then the translation would be meaning-preserving. A GitHub issue [5] has been submitted regarding removing this axiom, and this is currently being considered by the Creusot developers.

2.5.3 Ambiguous Translation (Incoherence)

An important property to consider when basing a translation on type derivations is whether the translation is *coherent*. A translation is coherent if all valid type derivations of the same expression translate to equivalent results. Unfortunately, our translation from Prusti specifications to Creusot specifications is not coherent.

When dereferencing a mutable reference with a lifetime that expires in the

```
#[after_expiry(**x == *result)]
#[after_expiry(**x == *old(*x))]
#[after_expiry(*old(*x) == *result)]
fn deref_mut<'a>(x: &'a mut &'a mut u32) -> &'a mut u32 {
  &mut *x
}
```

Translation 1:

```
#[ensures(*^x == ^result)] //Verifies
#[ensures(*^x == ^*x)] //Fails
#[ensures(^*x == ^result)] //Fails
fn deref_mut<'a>(x: &'a mut &'a mut u32) -> &'a mut u32;
```

Translation 2:

```
#[ensures(^^x == ^result)] // Fails
#[ensures(^^x == ^*x)] //Verifies
#[ensures(^*x == ^result)] //Fails
fn deref_mut<'a>(x: &'a mut &'a mut u32) -> &'a mut u32;
```

Figure 2.37: Ambiguous Translation Example

current state it can be ambiguous whether to type this dereference using the (*mut-curr-deref*) or the (*mut-fin-deref*) rules. This decision affects whether the dereference is translated into a $*$ (current) or ^ (final) operator. This situation can be caused by nested references that share a lifetime. For example, in Figure 2.37, `x` must have type Zombie⟨&'a mut &'a mut u32⟩ since it is not `Copy` but is moved from the prestate to the state where `'a` expires. This forces `*x` to be typed using (*mut-fin-deref*) which is translated to `^x`. `*x` has type &'a mut u32 which allows `**x` to be typed using (*mut-cur-deref*) which is translated to `*^x`. Alternatively, since `*x` has lifetime `'a`, `**x` can be typed by using (*sub*) to coerce `*x` into a Zombie⟨&'a mut u32⟩ and then using (*mut-fin-deref*) which is translated to `^^x`.

Unfortunately, while it seems reasonable that these two translations should be equivalent, Creusot is not currently able to determine this, so it is only able to verify certain specifications using one translation or the other. For example, Creusot is only able to prove the specification on line 1 when `**x` is translated to `*^x`, and only able to prove the specification on line 2 when `**x` is translated to `^^x`. This ambiguity can even cause specifications that are not themselves ambiguous to fail

```
#[after_expiry(**x == *result)]
#[after_expiry<'b>(*at::<'a>(*x) == *old(*x))]
fn deref_mut<'a, 'b>(x: &'a mut &'b mut u32)
  -> &'a mut u32 {
  &mut *x
}
```

Translation:

```
#[ensures(*^x == ^result)]
#[ensures(^^x == ^*x)]
fn deref_mut<'a, 'b>(x: &'a mut &'b mut u32)
  -> &'a mut u32;
```

Figure 2.38: Resolved Ambiguous Example

because their correctness depends on the ambiguous specifications being considered to be equal. For example, Creusot is unable to prove the specification on line 3, and it does not contain the ambiguous expression `**x`, but if Creusot assumes that the two possible translations of `**x` (`*^x` and `^^x`) were equal then it would be able to prove the specification.

Luckily this ambiguity should not occur very often in practice, since it relies on nested references that share a lifetime. Functions involving these references could often be generalized so that the lifetimes of the nested reference are different which would resolve the ambiguity, see Figure 2.38. The Creusot specifications from Figure 2.38 could also be directly written for the version of the function where the lifetimes are the same (Figure 2.37), by treating the lifetimes as though they were different. This idea, of treating the lifetimes as different, could be used when directly writing Creusot specifications where there are nested references that share a lifetime.

2.6 Summary

In this chapter, we explored the possibility of a direct run-time interpretation of Prusti specifications and considered having Rust type check this run-time interpretation to check if Prusti specifications are well-defined. We then showed that using this method does not work well, since it would severely limit the things that

Prusti Specification:

```
#[requires(f |= |x2: X| { requires(x2 == outer(x)) })]
#[ensures(old(f) (i) ~~> { result == outer(result) }))]
fn call<X>(f: impl FnOnce(X) -> Y, x: X) -> Y {
  f(x)
}
```

Creusot Specifications:

```
#[requires(f.precondition((x,))]
#[ensures(f.postcondition_once((x,), result)]
fn call<X>(f: impl FnOnce(X) -> Y, x: X) -> Y {
  f(x)
}
```

Figure 2.39: Prusti and Cresuot Closure Specification

could be specified about mutable references. We instead defined our own type system, which better handles mutable references. Finally, we used our type system to define a translation from well-typed Prusti specifications to Creusot specifications.

2.7 Future Work

Prusti [15] and Creusot [3] have both also added support for writing specifications that describe passed-in closures. The specification syntax, however, is quite different. Figure 2.39 shows the same function with both Prusti and Creusot specifications added to verify the function. Our current type system and translation could be extended to support translating these kinds of specifications as future work.

In Section 2.5.3, we described an ambiguity in the translation, where a Prusti dereference of a mutable reference could be translated to either Creusot's current or final operator. Future work could be done to investigate whether in these cases the two translations could soundly be assumed to be equivalent and if so, if Creusot could infer where to include these assumptions. This would likely have an impact on the useability of directly writing Creusot specifications in tricky cases.

In addition, Prusti is undergoing a substantial redesign of its code base, and our type system could be used to inform the new design on how it could better support mutable references in general.

Chapter 3

Ghost Unsoundness

Creusot's soundness [4] relies on the fact that each type that it uses has an *ownership predicate* in RustHornBelt [13]. When adding a new type to Creusot, we need to add the type's ownership predicate and prove the soundness of the type's API with respect to this ownership predicate in RustHornBelt. Proving an API sound in RustHornBelt gives confidence that the API can be soundly added to Creusot, however, there is no direct link between RustHornBelt and Creusot, so just because something is proven sound in RustHornBelt does not necessarily guarantee it would be sound to use it in Creusot. When Creusot introduced the `Ghost<T>` type Creusot did not give `Ghost<T>` an ownership predicate and `Ghost<T>` was subsequently found to be unsound. Unfortunately, using RustHornBelt's current *prophecy model*, creating a reasonable ownership predicate for `Ghost<T>` would not be possible, which makes proving the soundness of any part of `Ghost<T>`'s API challenging. In this chapter, we first give some background to RustHornBelt and its current prophecy model. We then explain the nature of the unsoundness. Finally, we describe changes to RustHornBelt's prophecy model which allow us to add an ownership predicate for `Ghost<T>` in order to prove the soundness, in RustHornBelt, of most of the properties that Creusot's `Ghost<T>` had.

Type	Ownership Predicate
`usize`	The physical value is an integer, and the logical value is the same integer
`Box<usize>`	The physical value is a pointer, that pointer points to an integer, and the logical value is the same integer as the one being pointed to
`Box<T>`	The physical value is a pointer, and that pointer points to some data that satisfies `T`'s ownership predicate with the same logical value
`(usize,usize)`	The physical value is a pair of two integers, and the logical value is a pair of the same two integers
`(T, U)`	The physical value is the pair (pt, pu), and the logical value is the pair (lt, lu) where pt satisfies `T`'s ownership predicate with lt, and pu satisfies `U`'s ownership predicate with lu

Table 3.1: Ownership predicates for various types

3.1 Background

RustBelt [9] is a formalization of Rust in Coq [2] using the Iris [10] verification framework, that can be used to prove that Rust APIs, that rely on unsafe code internally, are still safe to use. RustHornBelt is an extension of RustBelt that allows proving functional specifications. It includes proofs of specifications for some of Rust's standard library functions and primitive operations. In RustHornBelt each type has an ownership predicate, which formalizes the capabilities of each type. Each instance of a type can be thought of as having a *physical value* and a *logical value*. The physical value is the value that exists in memory at run time, while the logical value is the value that is referred to inside specifications. The ownership predicate is a function that takes in both the physical and logical values and returns whether the physical value is a valid instance of the type and whether the physical value can be accurately described by the logical value. Table 3.1 describes some examples of ownership predicates for some common types. In this thesis, we call an instance of a type valid if it satisfies its ownership predicate.

RustHornBelt also includes the concept of a prophecy model, introducing the idea of a *prophecy variable* which is used when handling mutable references.

```rust
/// Theoretical struct that represents
/// Rust's mutable references
struct Mut<T>{
  /// Current value
  /// (Creusot's * operator)
  c: T,
  /// Final value
  /// (Creusot's ^ operator)
  f: T
}

/// Theoretical function that creates a fresh
/// prophecy variable
fn fresh_pv<T>() -> T;

/// Theoretical function that resolves the prophecy
/// variable `pv` to `val`
fn resolve<T>(pv: T, val: T);
```

Figure 3.1: Definitions used for prophecy model interpretation

Prophecy variables are initially treated as uninterpreted, but are allowed to be *resolved* to a value later. After a prophecy variable has been resolved, it can be assumed to be equal to the value it was resolved to, for the purposes of verification. The logical value for a mutable reference is a pair of its *current value* and its *final value*. Mutably borrowing a variable creates a mutable reference whose current value equals the value of the variable that was borrowed, and whose final value is a fresh prophecy variable which conceptually represents the value that the variable will have when the borrow expires. When the mutable reference is dropped, its final value is resolved to its current value. In Figure 3.2 we give an example of how this prophecy model works. The first section of the figure shows some Rust code that uses mutable references. The second section shows an interpretation of this code using the definitions in Figure 3.1. The line numbers in this section correspond to the line numbers in the first section. The last section shows what the logical values of each variable are on each line, where `PV0` and `PV1` represent prophecy variables. On lines 2-3, `x` is mutably borrowed to create `y`, this creates the prophecy variable `PV0` which connects `y`'s final value to the value `x` will have

```
1 let mut x: (u32, u32) = (1,2);
2 let mut y: &mut (u32, u32) = &mut x;
3
4 let mut z: &mut u32 = &mut y.1;
5
6
7 *z = 3;
8 drop(z)
9 assert_eq!(x.1, 3);
```

```
let mut x = (1,2);
let mut y = Mut{c: x, f: fresh_pv()};
x = y.f;//When we next use x it will have y's final value
let mut z = Mut{c: y.c.1, f: fresh_pv()};
y.c.1 = z.f;
resolve(y.f, y.c);
z.c = 3;
resolve(z.f, z.x);
assert_eq!(x.1, 3);
```

line	x	y	z
1	(1,2)		
2	(1,2)	Mut{c:(1,2),f:PV0}	
3	PV0	Mut{c:(1,2),f:PV0}	
4	PV0	Mut{c:(1,2),f:PV0}	Mut{c:2,f:PV1}
5	PV0	Mut{c:(1,PV1),f:PV0}	Mut{c:2,f:PV1}
6	(1,PV1)		Mut{c:2,f:PV1}
7	(1,PV1)		Mut{c:3,f:PV1}
8	(1,3)		
9	(1,3)		

Figure 3.2: Example of Prophecy Model

when it expires. On lines 4-6, `y` is projected to create `z` which creates another prophecy variable `PV1` and resolves `y`'s final value `PV0` to `(1, PV1)` since `y`'s value will depend on `z` when it expires. On line 7, `z` is assigned `3`, which affects its current value. On line 8, `z` is dropped so its final value `PV1` is resolved to its current value `3`, since it can no longer be mutated. On line 9, we assert that `x.1` equals `3` which holds since `PV0` was resolved to `(1, PV1)` and `PV1` was resolved to `3`.

line	x logical value	x status	PV0	PV1
1	`(1,2)`	not blocked		
2	`(1,2)`	becoming blocked	unresolved	
3	`PV0`	blocked	unresolved	
4	`PV0`	blocked	unresolved	unresolved
5	`PV0`	blocked	unresolved	unresolved
6	`PV0`	blocked	resolved `(1,PV1)`	unresolved
7	`PV0`	blocked	resolved `(1,PV1)`	unresolved
8	`PV0`	blocked	resolved `(1,PV1)`	resolved `3`
9	`(1,3)`	not blocked	resolved `(1,PV1)`	resolved `3`

Figure 3.3: Example of Prophecy Model from RustHornBelt's perspective

3.2 Current RustHornBelt Prophecy Model

In this section, we describe RustHornBelt's prophecy model in more detail and describe the rules it has to prevent prophecy variable resolutions from leading to unsoundness.

In Figure 3.2 we showed how the prophecy model works from the perspective of the specifications, but from RustHornBelt's perspective, things are more complicated. From the specification's perspective, after a prophecy variable has been resolved, the prophecy variable acts as if it had always been fixed to the value it was resolved to, i.e., the prophecy variable acts as if it is no longer a prophecy variable (see Figure 3.2). Internally, however, from RustHornBelt's perspective, the prophecy variable is treated as being *equivalent* but not *identical* to the value the prophecy variable was resolved to and for the purposes of resolving other prophecy variables the resolved prophecy variable is still considered to exist, and to be marked as resolved. Figure 3.3 shows what is happening in Figure 3.2 from RustHornBelt's point of view. On line 6, `x`'s logical value is still `PV0` after it has been resolved but can be considered equivalent to `(1, PV1)`.

Additionally, in RustHornBelt, blocked variables are treated differently from non-blocked variables. When a variable is mutably borrowed, it is considered to be blocked, until the mutable reference expires. In Figure 3.2 `x` is considered blocked on lines 3-7 as shown in Figure 3.3. In RustHornBelt the ownership predicate for blocked values is different from the regular ownership predicate for that type. The

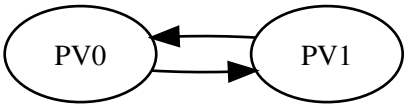


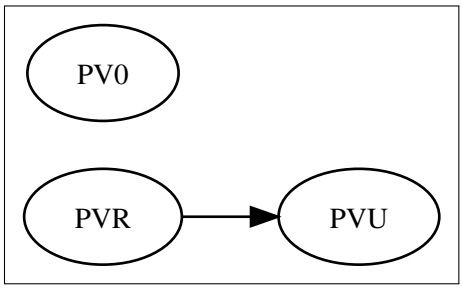
Figure 3.4: Cyclic Resolution

ownership predicate for the blocked value is: after the lifetime expires, the regular ownership predicate for the same type will be satisfied using the same physical value and an equivalent logical value, (with the meaning of equivalent as above). In particular, when a value becomes unblocked the value switches to having a different but equivalent logical value. For example, in Figure 3.3, on line 9 when `x` becomes unblocked its logical value switches from `PV0` to `(1,3)` so it no longer depends on any resolved prophecy variables.

We consider the *prophetic dependencies* of a value $\hat{a}$ to be all the prophecy variables that $\hat{a}$ depends on. When considering the various prophecy variable resolutions, we model them as a directed graph. The nodes are the prophecy variables and each prophecy variable that has been resolved to a value $\hat{a}$, has edges to $\hat{a}$'s prophetic dependencies.

Cycles in this graph can cause the resolutions to lead to unsoundness. For example, if `PV0` was resolved to `PV1` and `PV1` was resolved to `!PV0` we would have assumed that `PV0 == PV1 && PV1 == !PV0` which is contradictory. Figure 3.4 shows the graph corresponding to these resolutions, which has a cycle.

To avoid this kind of circular reasoning, RustHornBelt's prophecy model required that when resolving a prophecy variable x to a specific value $\hat{a}$, the *prophetic dependencies* of $\hat{a}$ must be unresolved; a core part of the soundness proof of RustHornBelt then amounts to showing that, and why this requirement is always satisfied. While we are trying to resolve x, we can no longer consider it to be unresolved, so this requirement implies that $\hat{a}$ cannot depend on x. Figure 3.5 shows an example graph of the existing resolutions of some prophecy variables. The table shows some examples of allowable resolutions, where `PV0` is the prophecy variable being considered, `PVR` is another prophecy variable that has been resolved, and `PVU` is another prophecy variable that is unresolved. Since dropping a mutable reference resolves its final value to its current value, dropping a mutable



x	$\hat{a}$	Resolution Allowed
PV0	PV0	No
PV0	(1, PV0.1)	No
PV0	(1, 2)	Yes
PV0	(1, PVR)	No
PV0	(1, PVU)	Yes
PV0	(PVR, PVU)	No

Figure 3.5: Allowable Resolution

reference requires that the dependencies of the mutable reference's current value must be unresolved. And since Rust lets users take mutable references to arbitrary owned types, RustHornBelt has a requirement that any valid type's logical value's prophetic dependencies are unresolved [1].

3.3 Ghost Type Unsoundness

Creusot and Verus currently have a `Ghost<T>` type for tracking and transporting logical values that are used for verification, but get erased at run time. `Ghost<T>` is represented as a Zero-Sized Type (ZST) so `Ghost<T>` does not add any overhead. Since a `Ghost<T>` type only represents a value, `Ghost<T>` is allowed to be freely copied. `Ghost<T>` can be created either by snapshotting a variable without consuming the variable or by mapping existing ghost types using a specification function (functions that are used in specifications). In Creusot, this is done using *ghost blocks* written as `gh!(...)`. A ghost block can contain a variable, in which case the block creates a ghost of this variable but does not consume the variable.

[1] https://gitlab.mpi-sws.org/iris/lambda-rust/-/blob/masters/rusthornbelt/theories/typing/type.v#L61

For example, given an `x` of type `T`, `gh!(x)` would have type `Ghost<T>`, and `x` would not be consumed. A ghost block can also contain a specification function, in which case the block conceptually converts the specification function into a ghost form where all the argument types and the return type a converted to ghost types, then recursively applies ghost blocks to its arguments and then calls the ghost form of the specification function.

For example:

```
1  #[logic]
2  fn ite<T>(i: bool, t: T, e: T) -> T {
3    if i {t} else {e}
4  }
5
6  fn test() {
7    let b: bool = true;
8    let x: u32 = 3;
9    let y: u32 = 5;
10   let z: Ghost<u32> = gh!(ite(b,x,y));
11   proof_assert!(*z == 3u32);
12 }
```

On line 10 `gh!` is used to create `z`, a `Ghost<u32>` whose value is `ite(b,x,y)` which equals 3. On line 11 `proof_assert!` is used to verify that `z`'s value is 3, which succeeds. Creusot's `proof_assert!` feature allows specifications to be checked inside of functions, and in this context, the * operator extracts the inner value of a Ghost.

In summary, Creusot's original `Ghost<T>` type had the following properties:

- `Ghost<T>` implements `Copy`
- `Ghost<T>` can be created from a `T` without consuming the `T`
- `Ghost<T>` can be created using a specification function (including the ^ operator)
- `Ghost<T>` can be converted back into a `T`, inside of a specification

Creusot's original implementation of `Ghost<T>` had an unsoundness reported by Jacques-Henri Jourdan [8] that allowed the tool to prove false by setting some

```
1  #[logic]
2  fn neg_prophecy(x: &mut Ghost<bool>) -> bool {
3      pearlite!{ ! *(^x).g }
4  }
5
6  #[ensures(false)]
7  pub fn f() {
8      let mut g1 = gh!(true);
9      let mut m: &mut Ghost<bool> = &mut s;
10     let g2 = gh!(neg_prophecy(b));
11     // snapshots b and then maps it with neg_prophecy
12     *m = g2;
13     drop(m)
14 }
```

line	g1	m	g2
8	true		
9	PV0	Mut{c:true,f:PV0}	
10	PV0	Mut{c:true,f:PV0}	!PV0
12	PV0	Mut{c:!PV0,f:PV0}	!PV0

Figure 3.6: Ghost unsoundness

ghost place to a value determined by looking at that place's own future value. Figure 3.6 is an example of this that contains a function that sets the value of a mutable `Ghost<bool>` to the negation of its own final value, and then drops the reference. This causes Creusot to resolve the final value to the current value. Since the current value is the negation of the final value, treating the current value as equivalent to the final value allows the tool to verify the postcondition `false`. When this program is compiled after verification, the `Ghost` operations are all treated as no-ops so the function returns successfully (see Figure 3.7) which is unsound since its postcondition is `false`.

While investigating this unsoundness, we also discovered an additional related unsoundness that is caused by using recursive ghost types. Figure 3.8 is an example of this, which creates an Algebraic Data Type (ADT) that contains itself (as seen on line 14), which breaks Creusot's assumption that all ADTs are inductively defined.

The root of both of these unsoundnesses is that we resolved a prophecy variable to another value that depended on that prophecy variable, creating a circular

```rust
pub fn f() {
    let mut g1 = ();
    let mut m = &mut g1;
    let g2 = ();
    *m = g2;
    drop(m);
}
```

Figure 3.7: Ghost unsoundness erased

```rust
1  enum Bad<'a> {
2      B,
3      R(Ghost<&'a mut Bad<'a>>)
4  }
5
6  fn test_bad() {
7      let mut x = Bad::B;
8      let m: &mut Bad<'_> = &mut x;
9      let g: Ghost<&mut Bad<'_>> = gh!(m);
10     *m = Bad::R(g);
11     proof_assert!(*m == Bad::R(g));
12     proof_assert!(^(*g) == ^m);
13     let _ = m;
14     proof_assert!(^(*g) == Bad::R(g));
15 }
```

line	x	m	g
7	B		
8	PV	Mut{c:B,f:PV}	
9	PV	Mut{c:B,f:PV}	Mut{c:B,f:PV}
10	PV	Mut{c:R(Mut{c:B,f:PV}),f:PV}	Mut{c:B,f:PV}

Figure 3.8: Ghost unsound cycle

dependency. As described in Section 3.2, this kind of unsoundness is prevented in RustHornBelt by requiring that any valid type's logical value's prophetic dependencies are unresolved. The type `Ghost<&mut T>` fails this requirement since it can continue to exist after the mutable reference it was a snapshot of was dropped. This drop causes the final value of the mutable reference to be resolved, at which point the `Ghost<&mut T>` depends on a resolved prophecy variable.

Prophectic Levels for Types

$$\text{level}(\texttt{usize}) = 0 \qquad \text{level}(\texttt{Box<T>}) = \text{level}(\texttt{T})$$

$$\text{level}(\texttt{(T, U)}) = \max(\text{level}(\texttt{T}), \text{level}(\texttt{U})) \qquad \text{level}(\texttt{Ghost<T>}) = 1 + \text{level}(\texttt{T})$$

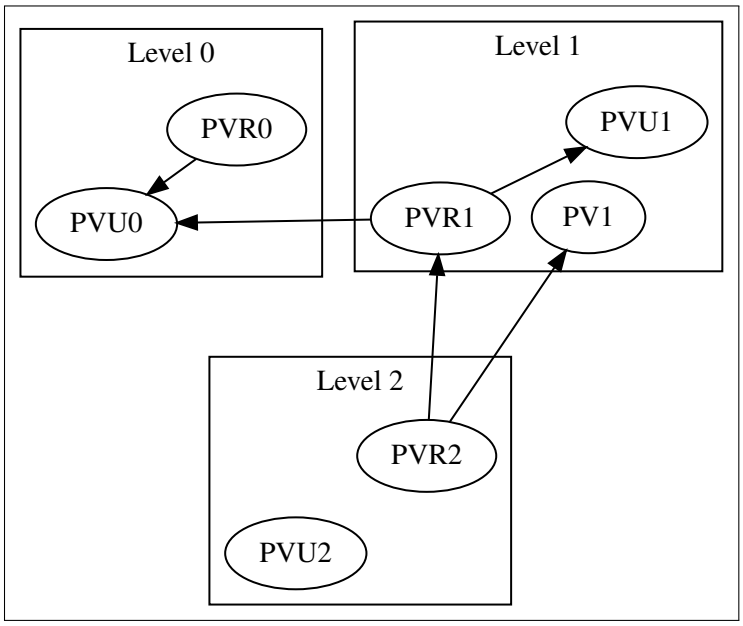
Figure 3.9: Examples of prophetic levels for types

Therefore, RustHornBelt in its current form would not allow ghosts of mutable references at all, and therefore cannot be used to prove any version of Creusot's `Ghost<&mut T>` type.

3.4 Solution/Formalization

As we have just shown, Creusot's original implementation of Ghost has proven to be unsound. Given that having a Ghost type is useful for verification, it is important to prove as many of the Ghost type's properties to be sound as possible. Unfortunately, the current RustHornBelt resolution rules, described in Section 3.2, are too restrictive to allow `Ghost<&mut T>` to exist. In this section, we modify RustHornBelt, so that it can be used to prove that most of `Ghost<T>`'s original properties are sound.

The goal of this modification is to make RustHornBelt's current resolution rules less restrictive by allowing prophecy variables to be resolved in terms of some values whose prophetic dependencies have been resolved, but still maintain their soundness by avoiding cycles. We assign each prophecy variable a *level* based on the type of the variable that was mutably borrowed to create the prophecy variable. The level for most types is the maximum of the levels of its type arguments, or 0 if the type does not have any type arguments. The exception to this is that `Ghost<T>`'s level is `T`'s level + 1 (see Figure 3.9). We now change the requirements made on the prophetic dependencies of $\hat{a}$, when we resolve a prophecy variable x to $\hat{a}$. The dependencies are allowed to have been resolved if they are at a *lower level* than x, but are not allowed to be at a *higher level* even if they are unresolved. Figure 3.10 is similar to Figure 3.5, showing a graph of existing resolutions and examples of allowable resolutions, but includes the levels of each



x	$\hat{a}$	Resolution Allowed
PV1	PVR0	Yes
PV1	PVU0	Yes
PV1	PVR1	No
PV1	PVU1	Yes
PV1	PVR2	No
PV1	PVU2	No
PV1	PVU1 + PVR0	Yes
PV1	PVU0 + PVR1	No

Figure 3.10: Revised Allowable Resolution

prophecy variable in the graph. These changes allow us to modify RustHornBelt's requirement so that a valid type's logical value's prophetic dependencies can be at a lower level even when they are resolved, but that they cannot be at a higher level at all. `Ghost<&mut T>` is able to meet this modified requirement since `Ghost<T>` has a higher level than `T` (see Figure 3.9).

Having modified RustHornBelt's prophecy model, we can now attempt to define the ownership predicate for `Ghost<T>`. At this point, a reasonable choice for the ownership predicate for `Ghost<T>` would be: a `Ghost<T>`'s logical

value's prophetic dependencies are at or below T's level, which is strictly below Ghost<T>'s level. Since a T's logical value now cannot have prophetic dependencies above T's level, we could also use T's logical value as the logical value for a Ghost<T>. Since this ownership predicate for Ghost<T> does not depend on the state of Ghost<T>'s prophetic dependencies this ownership predicate would continue to hold no matter what is done with the original T. This would allow a Ghost<T> to be constructed out of a T without consuming the T, as well as allow Ghost<T> to be copyable, which are two of the properties we want.

One problem with this choice is, for example, this ownership predicate would not allow us to prove the soundness of pair-projecting a Ghost<(T, U)> to a Ghost<T>. The pair type (T, U) is defined to have its level be the maximum of T and U's levels (see Figure 3.9). Because of this, knowing a (T, U) had all of its dependencies at or below (T, U)'s level is not sufficient to show the pair's first projection (a T) has all its dependencies at or below T's level (since T's level might be less than (T, U)'s level), which would be necessary to own a Ghost<T>. For example, given an x of type Ghost<(usize, Ghost<usize>)>, whose logical value is (PV1, 3), where PV1 is at level 1, x.0 would be invalid. x would be valid since its logical value's prophetic dependencies are at or below (usize, Ghost<usize>)'s level which is 1. The pair projection x.0 would have the type Ghost<usize> and x.0's logical value would be PV1. x.0 would not be valid since its logical value's prophetic dependencies are not at or below usize's level which is 0. Since x.0 is not valid, this pair projection would be invalid.

To handle this problem we give each type a new property, the *prophecy predicate*, to more precisely define which prophetic dependencies of the type's logical value are at which level (see Table 3.2)

For technical reasons related to RustHornBelt, this prophecy predicate also takes a list of prophecy variables as an additional parameter. For most types, this list just gets passed through recursively. For x: &mut T, the prophecy predicate also asserts the prophetic dependencies of ^x are in this list. For x: Ghost<T>, the prophecy predicate ignores the passed-in list and asserts there exists another list that satisfies the prophecy predicate for T.

We now replace RustHornBelt's original requirement which states that any

Type	Prophecy Predicate
`x: usize`	x does not depend on any prophecy variables
`x: Box<T>`	x satisfies `T`'s property.
`x: (T, U)`	$x.0$ satisfies `T`'s property and $x.1$ satisfies `U`'s property
`x: &mut T`	$*x$ satisfies `T`'s property and $\hat{}x$ prophetic dependencies are at or below `T`'s level
`x: Ghost<T>`	x satisfies `T`'s property

Table 3.2: Prophecy predicates for various types

valid type's logical value's prophetic dependencies have not been resolved, with two new requirements. The first one states that any valid type's logical value satisfies its prophecy predicate with a list of unresolved prophecy variables. The second one states that any type's prophecy predicate implies that its logical value's prophetic dependencies all have lower levels, or have the same level and are in the list. Together these requirements state that any valid type's logical value's prophetic dependencies all have lower levels, or have the same level and are unresolved. This combined lemma matches the requirement from the original RustHornBelt, aside from using our new notion of level-based dependencies.

We now define the ownership predicate for `Ghost<T>` to be: that there exists a list that satisfies the prophecy predicate for `T`. Since a `T`'s logical value satisfies `T`'s prophecy predicate with some list of prophecy variables, we could also use `T`'s logical value as the logical value for a `Ghost<T>`. Since this ownership predicate for `Ghost<T>` still does not depend on the state of the prophecy variables, this ownership predicate would continue to hold no matter what is done with the original `T`. This allows a `Ghost<T>` to be constructed out of a `T` without consuming the `T`, and allows `Ghost<T>` to be copyable.

We also prove that `Ghost<T>` can be created with a ghost block using a specification function, if knowing that all the arguments of the specification function satisfy their prophecy predicates implies that the result satisfies its prophecy predicates. This property is notably not true for the ^ operator which implies we cannot prove that using the ^ operator in ghost blocks is sound in RustHornBelt. This corresponds to the original unsoundness found in Creusot.

We were able to prove this property for specification functions that use a rich

subset of operations. We define a *ghost specification function* to be a specification function that uses this subset of operations, which contains the following:

- `struct` constructor
- `enum` constructor
- `struct` pattern matching
- `enum` pattern matching
- `if` expression
- `struct` projection
- `Box` dereference
- shared reference dereference
- mutable reference current operator
- convert a `Ghost<T>` to a `T`
- any operator that only involves `int` and `bool`
- call other ghost specification functions

We prove that we can create a `Ghost<T>` using a ghost specification function. This is more limited than the original unsound version of `Ghost<T>` which was allowed to be created using an arbitrary specification function.

Another limitation of our change to RustHornBelt is that recursive types cannot contain themselves under a `Ghost` as this would force their ghost level to be strictly larger than itself. This corresponds with the unsoundness demonstrated earlier in Figure 3.8.

3.5 Summary

In this chapter, we demonstrated some unsoundness in Creusot's original version of `Ghost<T>` related to cyclic prophecy resolution. We then introduced changes to RustHornBelt that allow us to prove most of the properties of `Ghost<T>` in RustHornBelt. These changes included updating the prophecy model and giving each type a prophecy predicate which is used to define the ownership predicate for `Ghost<T>`. Creusot could make changes to address these unsoundnesses in line with what our work has proven sound in RustHornBelt and have its `Ghost<T>` have the following properties:

- `Ghost<T>` implements `Copy`
- `Ghost<T>` can be created from a `T` without consuming the `T`
- `Ghost<T>` can be created using a ghost specification function as defined above
- `Ghost<T>` can be converted back into a `T`, inside of specification

but has the following limitations:

- `Ghost<T>` cannot be created using other specification functions such as the ^ operator
- Creusot would not support recursive ghost types

Despite these limitations, the `Ghost<T>` is useful when defining specifications and is now proven sound. The full soundness proof can be found in this fork[2] of RustHornBelt.

3.6 Future Work

Currently, in RustHornBelt the logical value of a mutable reference is a pair of the current and final values. This causes the equality of two mutable references to depend on the final value which prevents equality from being used in ghost functions. Future could be done to modify the logical value of mutable references to include a unique identifier of the prophecy variable. This change would make mutable references only equal if their final values are identical instead of just equivalent. This work could also include attempting to prove that this version of equality would be valid to be used in ghost functions.

[2] https://github.com/dewert99/rust-horn-belt/tree/thesis

Chapter 4

GhostPtrToken

4.1 Problem

Rust's aliasing XOR mutability discipline works well for tree-shaped data structures, but it does not work as well for data structures with intentional aliasing (e.g. doubly linked lists, skip lists, DAGs, and more general graphs). Rust has safe types (e.g. RefCell, Mutex, etc.) built into its standard library, that allow programmers to work around this restriction by allowing for controlled mutation to shared references of these types (this is known as interior mutability). This built-in mutation, however, often guarantees safety via run-time overhead such as run-time checking reference counts. An alternative strategy for creating this kind of data structure is to manually use raw pointers (which can freely alias memory) using unsafe Rust, but this loses Rust's inherent safety guarantees. Using unsafe Rust does not make the program unsafe but shifts the responsibility of proving safety from Rust's type checker to the developer or other tools.

If a developer wants to develop a data structure with intentional aliasing, they may want to avoid run-time overhead while retaining memory safety guarantees. A developer may also want to develop this kind of data structure with guaranteed functional properties that can be verified for correctness. Unfortunately, suitable verification tools for Rust (like Prusti [1] and Creusot [4]) cannot generally formally verify this correctness for data structures that rely on mutable aliasing. Verus [11] does support verifying some programs that have mutable aliases, but is limited

since it does not support functions that return mutable references.

In this chapter, we introduce `GhostPtrToken`, a Rust type that allows developers to verify the safety and functional correctness of an important class of Rust programs using raw pointers for mutable aliasing. This chapter describes the details of the API and provides some examples of how the API can be used.

4.2 Introduction

`GhostPtrToken` is a Rust type whose API allows it to manage a dynamic collection of raw pointers, which can be helpful when trying to implement a graph-like data structure. This API's methods include specifications to enforce that they are used safely. These specifications are designed to be compatible with existing verification techniques for safe Rust, such as those used in Prusti [1] or Creusot [4]. Rather than relying on run-time checks (or developer responsibility) for safety, using `GhostPtrToken` is safe, when its usage is verified by a verification tool. While `GhostPtrToken` is implemented as a Zero-Sized Type (ZST) and therefore has no run-time overhead, `GhostPtrToken` can be thought of as owning a collection of locations in memory and using raw pointers to index into that collection. Even though `GhostPtrToken` does not have any fields (only methods) and can be erased at run time, `GhostPtrToken` is still required to be a type (instead of a set of static functions), so that its method's specifications can refer to an individual instance instead of the global state. `GhostPtrToken` supports operations that allow adding and removing data as well as temporarily upgrading pointers to that data into shared or mutable references. `GhostPtrToken` also allows obtaining multiple mutable references to the data it owns at once but its specifications enforce that `GhostPtrToken` does not allow for the mutable references to alias.

4.3 API

In this section, we describe each method in the `GhostPtrToken` API. For each method, we include its specifications in both Creusot and Prusti styles, so the specifications can be more easily understood by readers who are more familiar with one tool or the other. The Prusti specifications could be translated to the Creusot

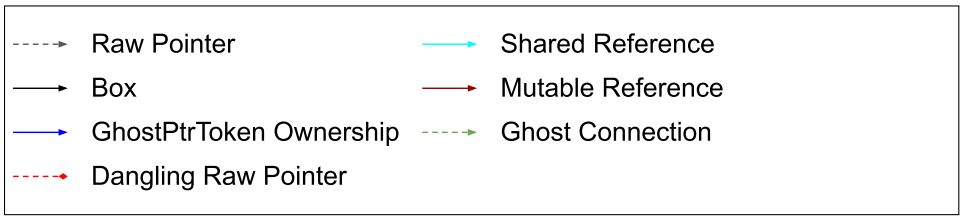


Figure 4.1: GhostPtrToken Diagram Legend

specifications by applying our translation from Chapter 2[1]. The specifications include the model (`@`) operator, which desugars to a function call, that converts a `GhostPtrToken<T>` to an `FMap<*const T, T>`, a mathematical finite map that maps the pointers the map owns to the values they point to. The `FMap` type provides several convenient specification methods for operating on maps. Using the (`@`) operator lets specifications involving `GhostPtrToken` use these methods. When comparing two `FMap`s using snapshot equality, the snapshot operator would recursively snapshot each value in the map. In Creusot, this (`@`) operator already exists and desugars to the `shallow_model` specification function. This operator does not exist in Prusti, but when included in a specification it would be translated to Creusot's (`@`) operator when using our translation, or this operator could be replaced with a specification method when actually using Prusti.

This section includes several diagrams to help visualize what the methods do. Figure 4.1 is a legend for these diagrams.

Because `GhostPtrToken`s own memory but do not track this memory at run time, their destructor leaks any memory that they own when they go out of scope. However, this does not affect the safety of `GhostPtrToken`, because Rust's memory safety guarantees do not include leak freedom. Nonetheless, the API includes a `drop` method which consumes a `GhostPtrToken`, but checks (at verification time) that it does not own any memory, which can be used to prevent that `GhostPtrToken` from leaking memory.

Using Prusti specs:

```
#[requires(self@.is_empty())]
```

[1] When translating, we consider raw pointers to implement `Plain` (sec Section 2.3.5) since they are only allowed to be compared in specifications.

```
pub fn drop(self)
```

Using Creusot specs:

```
#[requires(self@.is_empty())]
pub fn drop(self)
```

The precondition enforces that the `GhostPtrToken` did not own any memory when the call was made.

4.3.1 Basic Operations

In this section, we describe the basic operations for `GhostPtrToken`, including methods for creating it, adding and removing owned data, and projecting shared and mutable references.

The first method in our API is `new`, which creates a new `GhostPtrToken` that does not own any data.

Using Prusti specs:

```
#[ensures(result@ == FMap::empty())]
pub fn new() -> Self
```

Using Creusot specs:

```
#[ensures(result@ == FMap::empty())]
pub fn new() -> Self
```

The postcondition states that the return value will have the empty map as its model, reflecting that it does not own any data.

The next operation in our API is `ptr_from_box`, which casts a box into a raw pointer and takes ownership of the box's memory. `ptr_from_box` is used for transferring ownership into a `GhostPtrToken`.

Using Prusti specs:

```
#[ensures(!old((*self)@.contains(result)))]
#[ensures((*self)@ ==
  old((*self)@.insert(result, *val)))]
pub fn ptr_from_box(&mut self, val: Box<T>) -> *const T
```

Using Creusot specs:

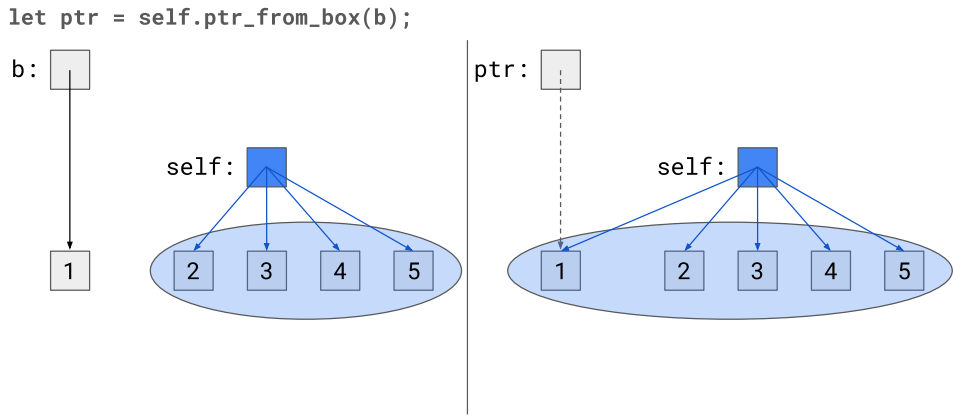


Figure 4.2: GhostPtrToken::ptr_from_box (A legend is given in Figure 4.1)

```
#[ensures(!(*self)@.contains(result))]
#[ensures((^self)@ == (*self)@.insert(result, *val))]
pub fn ptr_from_box(&mut self, b: Box<T>) -> *const T
```

The postconditions state that the `GhostPtrToken`'s model was updated to include the returned pointer mapped to the data in the box, and that it previously did not include that pointer.

The left side of Figure 4.2 shows a `GhostPtrToken` `self` that owns several locations in memory, as well as a `Box` `b` that owns the location it points to. Calling `ptr_from_box` consumes `b`, returns a pointer that points to the memory that used to be owned by `b`, and causes `self` to now own that location in memory.

The next operation is `ptr_to_box`, which casts a raw pointer into a box and returns ownership of the box's memory. `ptr_to_box` is used for transferring ownership out of a `GhostPtrToken`.

Using Prusti specs:

```
#[requires((*self)@.contains(ptr))]
#[ensures(*result == old((*self)@.lookup(ptr)))]
#[ensures((*self)@ == old((*self)@.remove(ptr)))]
pub fn ptr_to_box(&mut self, ptr: *const T) -> Box<T>
```

Using Creusot specs:

```
#[requires((*self)@.contains(ptr))]
```

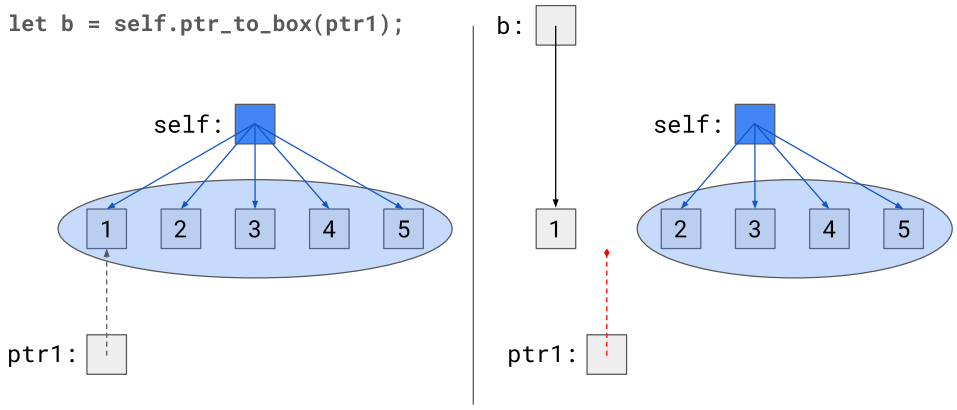


Figure 4.3: GhostPtrToken::ptr_to_box (see Figure 4.1)

```
#[ensures(*result == (*self)@.lookup(ptr))]
#[ensures((^self)@ == (*self)@.remove(ptr))]
pub fn ptr_to_box(&mut self, ptr: *const T) -> Box<T>
```

The precondition enforces that the `GhostPtrToken`'s model contains the passed-in pointer. The postconditions state that the `GhostPtrToken`'s model was updated to no longer include the passed-in pointer and that the data in the returned box matches the data that the pointer was previously mapped to in the `GhostPtrToken`'s model.

The left side of Figure 4.3 shows a `GhostPtrToken` `self` that owns several locations in memory and a raw pointer `ptr` that points to one of them. Calling `ptr_to_box` with `ptr` returns a `Box` that owns the memory that `ptr` used to be pointing to, and causes `self` to no longer own that location in memory.

The next operation is `ptr_as_ref`, which upgrades a raw pointer into a shared reference. `ptr_as_ref` is useful when you have a shared reference to a `GhostPtrToken` and want to access its inner data.

Using Prusti specs:

```
#[requires(self@.contains(ptr))]
#[ensures(*result == self@.lookup(ptr))]
pub fn ptr_as_ref(&self, ptr: *const T) -> &T
```

Using Creusot specs:

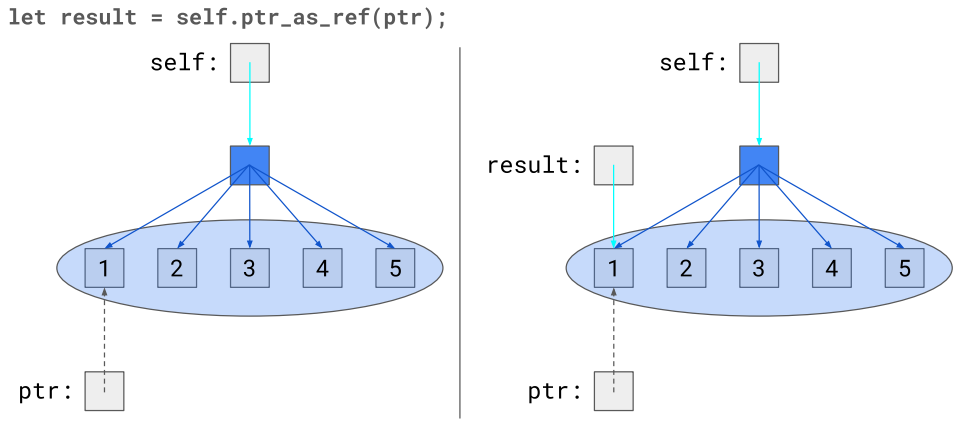


Figure 4.4: GhostPtrToken::ptr_as_ref (see Figure 4.1)

```
#[requires(self@.contains(ptr))]
#[ensures(*result == self@.lookup(ptr))]
pub fn ptr_as_ref(&self, ptr: *const T) -> &T
```

The precondition enforces that the `GhostPtrToken`'s model contains the passed-in pointer. The postcondition states that the data in the returned reference matches the data that the pointer is mapped to in the `GhostPtrToken`'s model.

The left side of Figure 4.4 shows a shared reference, `self`, to a `GhostPtrToken` that owns several locations in memory and a raw pointer `ptr` that points to one of them. Calling `ptr_as_ref` with `ptr` returns `result` which is a shared reference to the memory that `ptr` is pointing to. Since shared references implement `Copy`, `self` is still available to be used to get more shared references, including more references to the memory `ptr` is pointing to which would alias `result`.

The last of the basic operations is `ptr_as_mut` which upgrades a raw pointer into a mutable reference. `ptr_as_mut` can be used for modifying part of the data owned by the `GhostPtrToken`. Although this could also be done by extracting the data with `ptr_to_box`, modifying the box, and putting it back using `ptr_from_box`, this method is more convenient and idiomatic for a Rust API. This method is also needed to implement data structure methods that return mutable references.

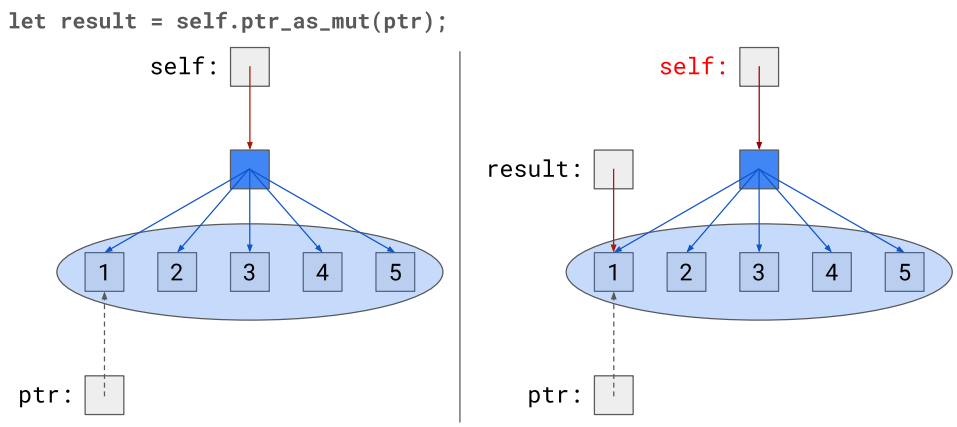


Figure 4.5: GhostPtrToken::ptr_as_mut (see Figure 4.1)

Using Prusti specs:

```
#[requires((*self)@.contains(ptr))]
#[ensures(*result == old((*self)@.lookup(ptr)))]
#[after_expiry((*self)@
  == old((*self)@.insert(ptr, *result)))]
pub fn ptr_as_mut(&mut self, ptr: *const T) -> &mut T
```

Using Creusot specs:

```
#[requires((*self)@.contains(ptr))]
#[ensures(*result == (*self)@.lookup(ptr))]
#[ensures((^self)@ == (*self)@.insert(ptr, ^result))]
pub fn ptr_as_mut(&mut self, ptr: *const T) -> &mut T
```

The precondition enforces the `GhostPtrToken`'s model contains the passed-in pointer. The postcondition states the data in the returned reference initially matches the data the pointer was previously mapped to in the `GhostPtrToken`'s model. The pledge states that after the returned reference expires, the `GhostPtrToken`'s model will be updated to have the pointer mapped to the reference's new value. Both of these hold since the returned mutable reference is an alias of the passed-in pointer.

The left side of Figure 4.5 shows a mutable reference, `self`, to a `GhostPtrToken` that owns several locations in memory, and a raw pointer `ptr`

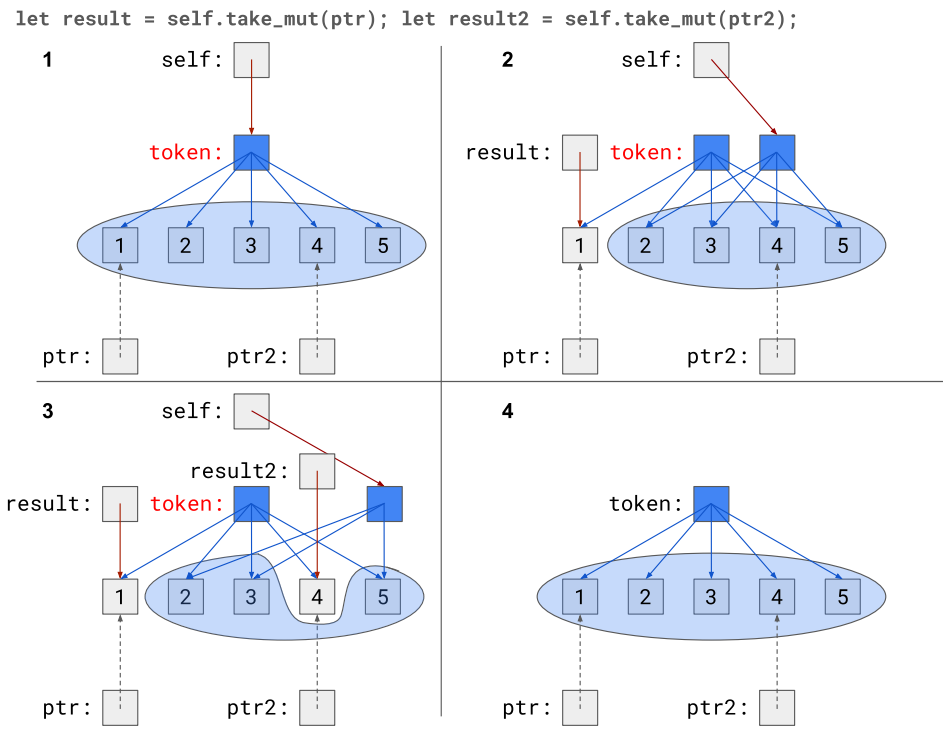


Figure 4.6: GhostPtrToken::take_mut (see Figure 4.1)

that points to one of them. Calling `ptr_as_mut` with `ptr` returns `result` which is a mutable reference to the memory that `ptr` is pointing to. At this point `self` is blocked until `result` expires, to prevent `self` from being used to create a mutable alias.

4.3.2 Advanced Operations

In this section, we introduce several additional operations to `GhostPtrToken` that add functionality to cover advanced use cases.

The first of these operations is `take_mut`; it is similar to `ptr_as_mut`, but it allows the `GhostPtrToken` to be used again after the call, without invalidating the returned mutable reference. `take_mut` can be used to get multiple non-aliasing mutable references to the data in the `GhostPtrToken` at once. This also makes `take_mut` useful for implementing mutable iterators.

Using Prusti specs:

```
#[requires((**self)@.contains(ptr))]
#[ensures(*result == old((**self)@.lookup(ptr)))]
#[ensures((**self)@ == old((**self)@.remove(ptr)))]
#[after_expiry((*old(*self))@
  == (*at<'o>(*self))@.insert(ptr, *result))]
// #[after_expiry(!(*at<'o>(*self))@.contains(ptr))]
// Note: this spec has not been proven
pub fn take_mut<'o, 'i>
  (self: &'o mut &'i mut GhostPtrToken<T>, ptr: *const T)
  -> &'i mut T
```

Using Creusot specs:

```
#[requires((**self)@.contains(ptr))]
#[ensures(*result == (**self)@.lookup(ptr))]
#[ensures((*^self)@ == (**self)@.remove(ptr))]
#[ensures((^*self)@ == (^^self)@.insert(ptr, ^result))]
// #[ensures(!(^^self)@.contains(ptr))]
// Note: this spec has not been proven
pub fn take_mut<'o, 'i>
  (self: &'o mut &'i mut GhostPtrToken<T>, ptr: *const T)
  -> &'i mut T
```

The precondition enforces that the `GhostPtrToken`'s model contains the passed-in pointer. The first postcondition states that the data in the returned reference initially matches the data that the pointer was previously mapped to in the `GhostPtrToken`'s model. The second postcondition states that after the call, the `GhostPtrToken`'s model was updated to no longer include the pointer, which prevents the `GhostPtrToken` from being used to create a mutable alias of the returned mutable reference. The first pledge states that when the returned mutable reference expires, the model of dereferencing in the prestate will be equal to the model of dereferencing in the poststate, with the pointer mapped to the value of the returned reference inserted. This is what guarantees that the `GhostPtrToken` that was originally borrowed, still owns the pointer when the returned reference expires. The second (commented out) pledge states that when the returned mutable reference expires, the model of dereferencing in the poststate will not contain

the pointer. This shows that the pointers returned by any calls to `ptr_from_box` that are made after calling this method will not be equal to the original passed-in pointer (unless they were subsequently removed). Unfortunately, this last pledge has not been formally proven (see Section 4.5).

Diagram 1 of Figure 4.6 shows a mutable reference to a `GhostPtrToken`, `self`, that owns several locations in memory, and shows two raw pointers, `ptr` and `ptr2`, that each point to a different memory location that is owned by the `GhostPtrToken`. Calling `take_mut` with `ptr` returns `result` which is a mutable reference to the memory that `ptr` is pointing to, and causes `self` to reference a smaller `GhostPtrToken` that does not own that location in memory. Then calling `take_mut` again, with `ptr2`, returns `result2` which is a mutable reference to the memory that `ptr2` is pointing to, and causes `self` to reference an even smaller `GhostPtrToken` that does not own either location in memory. If we had called `take_mut` again with `ptr1` instead of `ptr2`, this second call would not have verified since the `GhostPtrToken` that `self` was referencing did not own `ptr1`'s location. After the lifetime of these mutable references expires, the `GhostPtrToken` that was originally borrowed to create `self` still owns all of its original locations in memory.

As discussed earlier this method is similar to `ptr_as_mut`, and can also be used to implement it as follows:

```
pub fn ptr_as_mut(&mut self, ptr: *const T) -> &mut T {
    let mut t = self;
    t.take_mut(ptr)
}
```

The next operation is `merge`, which adds all the owned locations from one `GhostPtrToken` into another. It can be used for combining the data from several `GhostPtrTokens`.

Using Prusti specs:

```
#[ensures(old((*self)@.disjoint(other@)))]
#[ensures((*self)@ == old((*self)@.union(other@)))]
pub fn merge(&mut self, other: GhostPtrToken<T>)
```

Using Creusot specs:

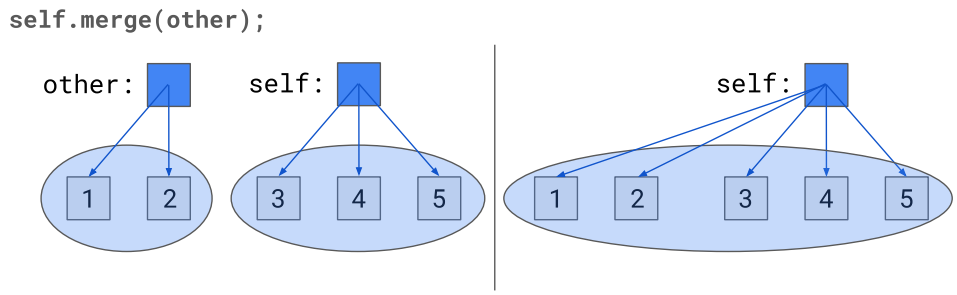


Figure 4.7: GhostPtrToken::merge (see Figure 4.1)

```
#[ensures((*self)@.disjoint(other@))]
#[ensures((^self)@ == (*self)@.union(other@))]
pub fn merge(&mut self, other: GhostPtrToken<T>)
```

The postconditions state that the original `GhostPtrToken`'s model was updated to include the data from the other `GhostPtrToken` and that it previously didn't include any of the pointers from the other `GhostPtrToken`.

The left side of Figure 4.7 shows two `GhostPtrTokens` `self` and `other` that each owns several locations in memory. Calling `merge` consumes `other`, and causes `self` to own all of the locations in memory.

The last operation is `shrink_token_ref`, which returns a shared reference to a `GhostPtrToken` whose model contains a subset of the pointers from the original. This method makes use of the `Ghost<T>` type, introduced in Section 3.3. This method might seem unnecessary since `self` would be able to do everything `result` could do. However, this method can be useful for simplifying verification proofs as will be demonstrated in the `Iter::next` method in Section 4.4.1.

Using Prusti specs:

```
#[requires(new_map.subset(self@))]
#[ensures((*result)@ == *new_map)]
pub fn shrink_token_ref
  (&self, new_map: Ghost<FMap<*const T, T>>)
  -> &GhostPtrToken<T>
```

Using Creusot specs:

```
#[requires(new_map.subset(self@))]
```

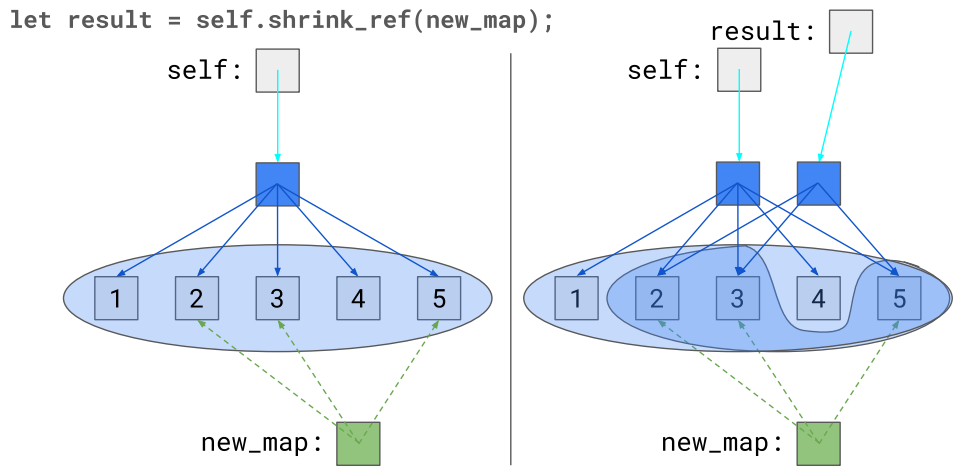


Figure 4.8: GhostPtrToken::shrink_token_ref (see Figure 4.1)

```
#[ensures((*result)@ == *new_map)]
pub fn shrink_token_ref
  (&self, new_map: Ghost<FMap<*const T, T>>)
  -> &GhostPtrToken<T>
```

The precondition enforces that the `GhostPtrToken` contains all the pointers in the desired map. The postcondition states that the returned `GhostPtrToken`'s model is this map.

The left side of Figure 4.8 shows a shared reference to a `GhostPtrToken`, `self`, which owns several locations in memory, as well as a `Ghost<FMap>` `new_map` which represents a subset of these locations. Calling the method `shrink_token_ref` with `new_map` returns `result` which is a shared reference to a `GhostPtrToken` that only owns this subset of the locations in memory.

4.4 Examples

In this section, we present two examples that use `GhostPtrToken` to implement API which internally requires mutable aliasing. The first example is a linked list with a tail pointer, and the second example is a memory allocator. For each example we present and discuss the code that makes use of the `GhostPtrToken` methods, highlighting how using `GhostPtrToken` can provide a way to access

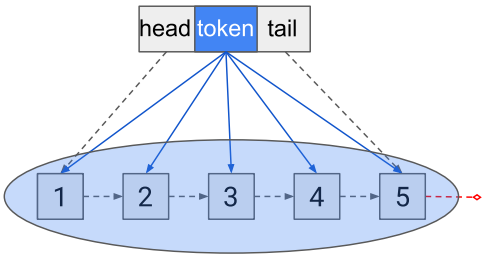


Figure 4.9: LinkedList

raw pointers that can be verified for safety and correctness.

The data structure's API methods often need preconditions in order the satisfy the preconditions of the `GhostPtrToken`'s methods. In order to, simplify these preconditions we define an invariant property (a property that should always hold true) that relates the other fields of the struct to the memory owned by the `GhostPtrToken`, and we always use this invariant as the precondition. Our data structure API methods, that return or mutate the data, also ensure this invariant, (after returning or mutating) so that the invariant holds for all instances of the data structure. The data structures we present in these examples have a predicate specification method called `invariant` to check if an instance satisfies the data structures' invariant. We consider instances to be *valid* when they satisfy their invariant.

A simple example of a type with an invariant is the `MyBox<T>` type which is meant to be a replacement for the `Box<T>` type. It has two fields, a raw pointer `ptr`, and a `GhostPtrToken<T>` `token`:

```
struct MyBox<T> {
  ptr: *const T,
  token: GhostPtrToken<T>,
}
```

Its invariant is that `token` owns the memory that `ptr` points to.

4.4.1 Linked List

The first example is implementing a linked list with a tail pointer (having a tail pointer allows two linked lists to be efficiently appended). Implementing this

linked list efficiently in Rust would be unsafe without using `GhostPtrToken` since the implementation involves aliasing (i.e., the tail pointer aliases some other node's next pointer). Figure 4.9 demonstrates this aliasing where both `tail` and node 4's `next` pointer both alias node 5. Using `GhostPtrToken` guarantees the memory safety of our implementation, as long the preconditions on the linked list's methods hold (i.e., the linked list satisfies its invariant).

The linked list is defined as a struct that contains a `head` pointer, that points to a `Node<T>`, a `tail` pointer that points to a `Node<T>` and a `GhostPtrToken` `token` of type `Node<T>`. The `GhostPtrToken` is used to manage the ownership of the memory of the nodes in the linked list.

```
struct Node<T> {
    data: T,
    next: *const Node<T>,
}

pub struct LinkedList<T> {
    head: *const Node<T>,
    tail: *const Node<T>,
    token: GhostPtrToken<Node<T>>,
}
```

The linked list satisfies its invariant if the `head` is null and the `token` is empty (i.e., an empty linked list) or, if repeatedly following each node's `next` pointer, starting with `head`, eventually leads to the `tail` pointer, and the `tail`'s pointer's next pointer is null, and all these nodes are in the `GhostPtrToken`, and it does not contain any other nodes. Figure 4.9 shows an example of a `LinkedList` that satisfies its invariant. `LinkedList` contains a specification method called `invariant` that returns whether the invariant is satisfied.

`LinkedList` also contains a specification method called `model` that returns a sequence of `T`'s (`Seq<T>`) that contains the values of the nodes (`T`), in order, starting at the head.

Our `LinkedList` has the basic functionality expected of a linked list, but we only highlight the methods that make use of the `GhostPtrToken` API.

The `new` method creates a new empty `LinkedList`.

```
#[ensures(result.invariant())]
#[ensures(result.model() == Seq::new())]
pub fn new() -> Self {
    LinkedList {
        head: ptr::null(),
        tail: ptr::null(),
        token: GhostPtrToken::new(),
    }
}
```

This method calls the `GhostPtrToken::new` function to create a new empty `GhostPtrToken`.

The `singleton` method creates a new `LinkedList` with a single element in it.

```
#[ensures(result.invariant())]
#[ensures(result.model() == Seq::singleton(v))]
pub fn singleton(v: T) -> Self {
    let mut token = GhostPtrToken::new();
    let node = Node {data: v, next: ptr::null()};
    let ptr = token.ptr_from_box(Box::new(node));
    LinkedList {head: ptr, tail: ptr, token}
}
```

This method also calls the `GhostPtrToken::new` function to create a new empty `GhostPtrToken`, but then calls `ptr_from_box` to add one node to the token.

The `dequeue` method removes the first element from the list and returns it, or returns `None` if there are no elements.

```
#[requires((*self).invariant())]
#[ensures((^self).invariant())]
#[ensures(match result {
    Some(val) => (*self).model() ==
        Seq::singleton(val).concat((^self).model()),
    None => ^self == *self && (*self)@ == Seq::new()
})]
pub fn dequeue(&mut self) -> Option<T> {
```

```
    if self.head.is_null() {
        None
    } else {
        let node = self.token.ptr_to_box(self.head);
        self.head = node.next;
        Some(node.data)
    }
}
```

This method uses `ptr_to_box` to extract the element from the token. If we were not using `GhostPtrToken` in the implementation of our linked list, the implementation would require using unsafe Rust since we would be turning a raw pointer into a box. This can be done safely, using `GhostPtrToken` because of the invariant.

The `append` method takes another linked list and appends it to the end of this linked list.

```
#[requires((*self).invariant())]
#[requires(other.invariant())]
#[ensures((^self).invariant())]
#[ensures((^self).model() ==
    (*self).model().concat(other.model()))]
pub fn append(&mut self, other: LinkedList<T>) {
    if self.head.is_null() {
        *self = other
    } else if !other.head.is_null() {
        let tail = self.token.ptr_as_mut(self.tail);
        tail.next = other.head;
        self.token.merge(other.token);
        self.tail = other.tail;
    }
}
```

This method first uses `ptr_as_mut` to get a mutable reference to `tail`. It then sets the `tail`'s next pointer to be the `head` of the other linked list, and then uses `merge`, to merge the other linked list's `GhostPtrToken` with this one's `GhostPtrToken`. Even though the `merge` method does not do anything at run time, calling it is required to verify the postconditions, since `self.token` is

required to own all of the nodes. Since this method casts a raw pointer to a mutable reference it would be unsafe if were not using `GhostPtrToken`. An `enqueue` method could be easily implemented by calling `append` with `singleton`.

Because iterating over a list is common, our `LinkedList` contains methods that return an iterator of either shared references (`iter`) or mutable references (`iter_mut`). The iterators are defined as structs that contain a `curr` pointer that points to a `Node<T>`, a shared or mutable reference respectively, to a `GhostPtrToken` of type `Node<T>` and a `tail` pointer that points to the `tail`. Because the iterator's `tail` pointer is never used at run time (it is used in the invariant) we store it in a `Ghost` (see Section 3.3), to avoid run-time overhead.

```
pub struct Iter<'a, T> {
    curr: *const Node<T>,
    token: &'a GhostPtrToken<Node<T>>,
    tail: Ghost<*const Node<T>>,
}

pub struct IterMut<'a, T> {
    curr: *const Node<T>,
    token: &'a mut GhostPtrToken<Node<T>>,
    tail: Ghost<*const Node<T>>,
}
```

As with the `LinkedList`, we define invariants for `Iter` and `IterMut`. Both satisfy their invariant, if the `LinkedList` created using `curr`, `token` and `tail`, satisfies the its invariant. Both `Iter` and `IterMut` also contain a specification method called `model` that returns the model of the `LinkedList` created using `curr`, `token` and `tail`. When an iterator is first created its `model` corresponds to the `model` of the linked list that was borrowed to create it.

Each iterator struct contains a `next` method that returns an optional reference to a `T`.

For `Iter`, `next` returns an optional shared reference:

```
#[requires((*self).invariant())]
#[ensures((^self).invariant())]
#[ensures(match result {
    Some(val) => (*self).model() ==
```

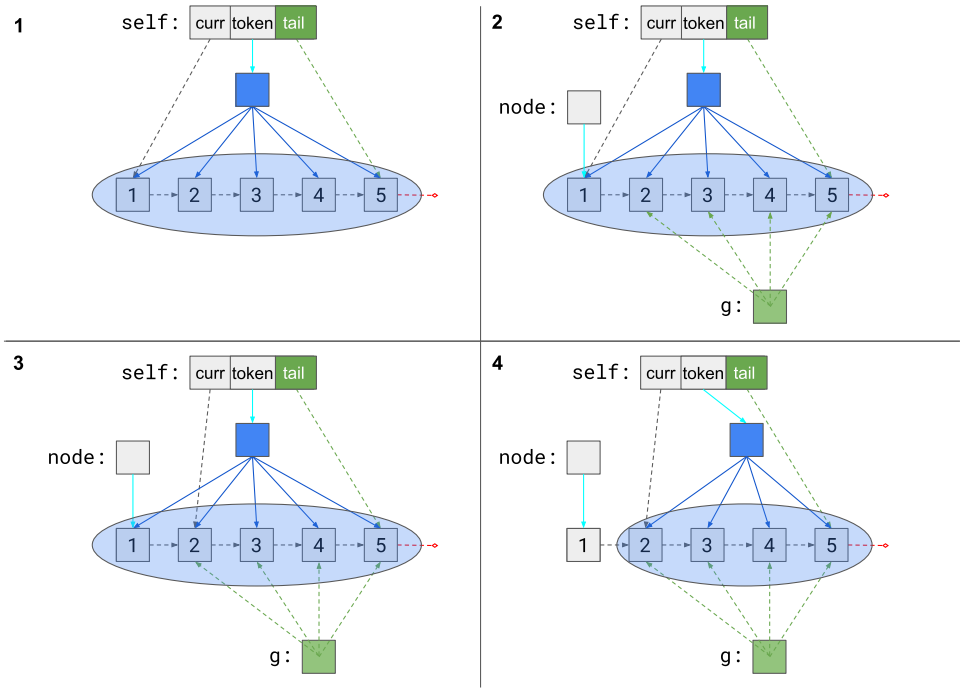


Figure 4.10: Iter::next

```
        Seq::singleton(*val).concat((^self).model()),
    None => ^self == *self && (*self).model()==Seq::new()
})]
fn next(&mut self) -> Option<&'a T> {
    if self.curr.is_null() {
        None
    } else {
        let node = self.token.ptr_as_ref(self.curr);
        let g = gh!(self.token@.remove(self.curr));
        self.curr = node.next;
        self.token = self.token.shrink_token_ref(g);
        Some(&node.data)
    }
}
```

Figure 4.10 shows the state of the `Iter` struct as the call to `next` progresses. At the start of the call, we have an `Iter` struct `self` which satisfies its invariant

(diagram 1). On line 12 we call `ptr_as_ref` to extract a shared reference `node` to the current node. On line 13 we create `g` a `Ghost<FMap<_>>` that points to all the nodes except the first one (diagram 2). We set `curr` to `node`'s `next` on line 14. At this point `curr` is pointing to the correct place but `Iter`'s invariant does not hold since the `token` still owns node 1 even though that node is not in the list starting at `curr` (diagram 3). We then call `shrink_token_ref` with `g`, to shrink `token` (the `&GhostPtrToken`) so it now satisfies the invariant again (diagram 4).

Since an `IterMut` can mutate the linked list that was borrowed to create it, `IterMut` has an additional predicate specification method, `fin_invariant`, that is used to ensure that this linked list will still be valid after its lifetime expires. This `fin_invariant` method returns whether the linked list created using `curr`, `token` and `tail`, when `token` expires, satisfies its invariant. Dropping a valid iterator resolves its `token` which allows us to learn that the iterator also satisfies its `fin_invariant`. Whenever we own a valid iterator `iter` that is borrowing from the linked list `list`, such that `iter.fin_invariant()` implies `list.invariant()`, we know that `list` is still valid since we could drop `iter` to learn `iter.fin_invariant()`. The link list's `iter_mut` method has a postcondition:

```
result.fin_invariant() ==> (^self).invariant()
```

which ensures the linked list `self` is still valid after the call. The iterator's `next` method has a postcondition:

```
(^self).fin_invaraint() ==> (*self).fin_invariant()
```

which ensures that if the linked list the iterator is borrowing from was valid before the call the linked list would still be valid afterwards. This ensures that the linked list is valid at all steps during the iteration. `IterMut` also contains another specification method, `fin_model` that returns the model of the linked list created using `curr`, `token` and `tail`, when `token` expires. When an iterator is first created its `fin_model` corresponds to the `model` the linked list will have after the iteration is complete.

For IterMut, `next` returns an optional mutable reference:

```
#[requires((*self).invariant())]
```

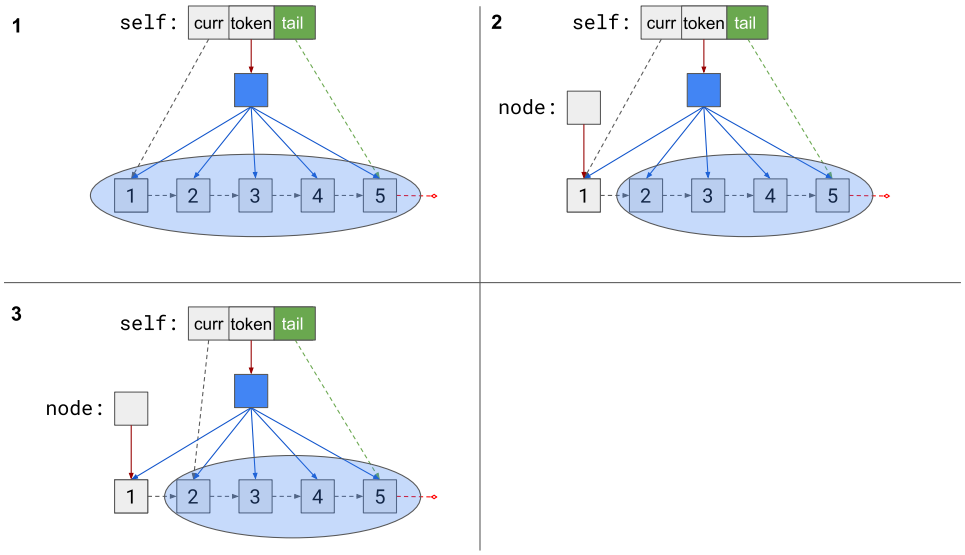


Figure 4.11: IterMut::next

```
#[ensures((^self).invariant())]
#[ensures(match result {
    Some(val) => (*self).model() ==
        Seq::singleton(*val).concat((^self).model()),
    None => ^self == *self && (*self).model()==Seq::new()
})]
#[ensures((^self).fin_invariant() ==>
    (*self).fin_invariant())]
#[ensures((^self).fin_invariant() ==> match result {
    Some(val) => (*self).fin_model() ==
        Seq::singleton(^val).concat((^self).fin_model()),
    None => true
})]
fn next(&mut self) -> Option<&'a mut T> {
    if self.curr.is_null() {
        None
    } else {
        let node = self.token.take_mut(self.curr);
        self.curr = node.next;
        let res = &mut node.data;
```

```
22            Some(res)
23        }
24    }
```

Figure 4.11 shows the state of the `IterMut` struct as the call to `next` progresses. Even though the specifications are more complex, the implementation of `next` for `IterMut` is simpler than for `Iter`. We first call `take_mut` to extract a mutable reference to the current node (diagram 2). We then simply set `curr` to node's `next` (diagram 3). Since, by design, `take_mut` already shrinks the token, the `IterMut` struct will already satisfy its invariant at this point.

4.4.2 Lazy Allocator/Deallocator

For our second example, we introduce a Lazy Memory Allocator/Deallocator that uses a `GhostPtrToken`. This lazy allocator allows users to consume a `Box`, converting the box into a mutable reference without leaking memory, which cannot currently be done in safe Rust. At a high level, the lazy allocator creates a scope within which mutable references can be created and then at the end of the scope the lazy allocator frees all the memory at once. In order to keep track of what memory to free later, the lazy allocator keeps a list of raw pointers which alias the mutable references it returns. Freeing raw pointers is normally unsafe, but can be done safely by using `GhostPtrToken`. One use case could be running a performance-sensitive operation that requires a number of mutable references, where for performance reasons the memory should not be freed until after the operation has been completed.

The lazy allocator is implemented using two structs, `LazyAllocator` and `LazyAllocatorData`. The `LazyAllocator` struct is part of the public API and the `LazyAllocatorData` struct is used as a helper as part of the implementation. A `LazyAllocator` is created using a mutable reference to a `LazyAllocatorData`.

The `LazyAllocatorData` struct contains a `GhostPtrToken` `token` of type `T` and a vector of pointers.

```
1 pub(super) struct LazyAllocatorData<T: ?Sized> {
2     token: GhostPtrToken<T>,
3     allocated: Vec<*const T>,
```

```
}
```

`LazyAllocatorData` satisfies its invariant if all the pointers that are in the vector are owned by `token` and the `token` owns no other pointers. The struct contains a `drop` method which uses `GhostPtrToken::ptr_to_box` to convert each pointer in the vector into a box then frees the box, and finally calls `GhostPtrToken::drop` to drop the token and prove we did not leak any memory.

```
#[requires(self.invariant())]
// ensures no memory leaks
fn drop(self) {
    let mut token = self.token;
    #[invariant(alloc_invariant(token@, iter@))]
    for ptr in self.allocated {
        let _ = token.ptr_to_box(ptr);
    }
    token.drop();
}
```

The second struct, `LazyAllocator` contains `token`, a mutable reference to a `GhostPtrToken` of type `T`, and `allocated`, a mutable reference to a vector of pointers.

```
pub struct LazyAllocator<'a, T: ?Sized> {
    token: &'a mut GhostPtrToken<T>,
    allocated: &'a mut Vec<*const T>,
}
```

A `LazyAllocator` satisfies its invariant, if its token is currently empty. `LazyAllocator` contains another specification method, `fin_invariant`, which is similar to what we had in `IterMut`. This method returns whether both the value `allocated` has when the lifetime `'a` expires is equal to `allocated`'s current value with some extra elements appended on, and `token` contains all these extra elements when `'a` expires.

The key method of `LazyAllocator` is `accept_box`, which takes in a Box and returns a mutable reference. Rust has a built-in function `Box::leak` that can also do this, but as the name suggests, it leaks the memory.

```
#[requires((*self).invariant())]
#[ensures((^self).invariant())]
#[ensures((^self).fin_invariant() ==>
  (*self).fin_invariant())]
#[ensures(*result == *memory)]
pub fn accept_box(&mut self, memory: Box<T>) ->
  &'a mut T {
    let ptr = self.token.ptr_from_box(memory);
    self.allocated.push(ptr);
    self.token.take_mut(ptr)
}
```

The `accept_box` method uses `GhostPtrToken::ptr_from_box` to create a pointer. The `accept_box` method then adds the pointer to the vector so the pointer's memory can be freed later. The `accept_box` method then calls `GhostPtrToken::take_mut` to convert the pointer to a mutable reference and then returns the mutable reference.

The `with_lazy_allocator` function is the public interface that is used to initially get access to a `LazyAllocator`. This function takes in another function, `f`, with a `LazyAllocator` as a parameter. This passed-in function `f` allows the user to allocate memory (by calling `accept_box` on the allocator) which will be freed after the function returns.

```
#[requires(forall<x: LazyAllocator<'_, T>> x.invariant()
  ==> f.precondition((x,))
    && (forall<u: U> f.postcondition_once((x,), u)
      ==> x.fin_invariant()))]
#[ensures(exists<x: LazyAllocator<'_, T>> x.invariant()
  && f.postcondition_once((x,), result))]
pub fn with_lazy_allocator<T, U, F>(f: F) -> U
    where F: FnOnce(LazyAllocator<'_, T>) -> U {
    let mut data = LazyAllocatorData::new();
    let allocator = LazyAllocator::new(&mut data);
    let res = f(allocator);
    data.drop();
    res
}
```

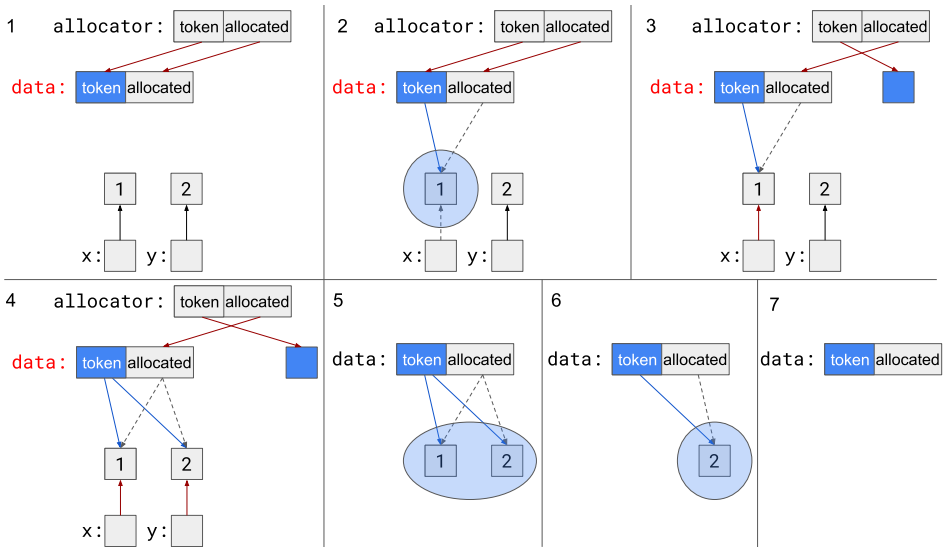


Figure 4.12: LazyAllocator

The `with_lazy_allocator` function creates a `LazyAllocatorData`, `data`, and mutably borrows `data` to create a `LazyAllocator`, `allocator`. The function then calls the passed-in function on the `allocator` and finally calls `drop` on `data`, which uses `GhostPtrToken`, to free all the memory that was allocated by `allocator` in the function.

The following shows an example of using `LazyAllocator`:

```
fn demo() {
    let x = Box::new(1);
    let y = Box::new(2);
    with_lazy_allocator(|mut allocator| {
        let x = allocator.accept_box(x);
        let y = allocator.accept_box(y);
    })
}
```

Figure 4.12 shows how this example works. Diagram 1 shows the state within `with_lazy_allocator` just before calling the passed-in function. The passed-in function calls `accept_box` on `x`. Diagram 2 shows the state within the call to `accept_box` after calling `ptr_from_box` and adding the pointer to be freed

later. Diagram 3 shows the state after `accept_box` calls `take_mut` and returns `x` as a mutable reference. Diagram 4 shows the state after calling `accept_box` again with `y`. Diagram 5 shows the state within `with_lazy_allocator` after the passed-in function returns and allocator expires. Diagram 6 shows the state within drop as the memory is being freed and finally diagram 7 shows the state when calling `GhostPtrToken::drop`.

4.5 Formalization

So far we have introduced the API and some examples of using `GhostPtrToken` to implement safe and efficient data structures that involve mutable aliasing. The safety comes from formal verification and relies on the correctness of the specifications of the `GhostPtrToken` API. In order to gain confidence that these specifications are correct in Creusot, we prove that the implementation and specifications are sound in RustHornBelt [13]. In this section, we first define the ownership predicate for `GhostPtrToken`. Unfortunately, because of limitations in RustHornBelt, using this ownership predicate we are unable to prove one of the specifications for the `take_mut` method. In order to work around this we introduce the concept of a `GhostPtrTokenMut` type that behaves the same as a `&mut GhostPtrToken`, but allows us to prove this specification.

As previously discussed in Chapter 3, to formalize what a new type means when it is added to Creusot, a corresponding ownership predicate should be added to RustHornBelt. `GhostPtrToken` is a ZST, so its ownership predicate only depends on its logical value, similar to the `Ghost` type. A `GhostPtrToken`'s logical value is a collection of pairs where the first element in each pair is a pointer. `GhostPtrToken<T>`'s ownership predicate states: for each pair $(ptr, lval)$, the ptr points to some data $pval$ where $pval$ satisfies `T`'s ownership predicate with $lval$ as the logical value, and we have the right to free this data.

Using this ownership predicate we can prove most of the specifications for `GhostPtrToken` but cannot prove the following specification on the `take_mut` method:

```
#[ensures(!(^^self)@.contains(ptr))
```

After calling `token.take_mut(ptr)` where `token` is a `&mut GhostPtrToken<T>` the problematic specification would imply:

```
!(^token).contains(ptr)
```

In order to be valid, according to the ownership predicate for mutable references defined in RustHornBelt, a mutable reference's final value must be an unresolved prophecy variable. Since `token` is a `&mut GhostPtrToken<T>`, `^token` must be an unresolved prophecy variable, and since unresolved prophecy variables are treated as uninterpreted, we could not have been able to learn that `!(^token).contains(ptr)`, and therefore could not have proven the specification. We can generalize this idea to say that we cannot prove things about the final value of a mutable reference a function produces.[2]

To solve this, we consider a new type, `GhostPtrTokenMut`, that acts like a mutable reference to a `GhostPtrToken` and use it to define our own ownership predicate, instead of using the ownership predicate for `&mut GhostPtrToken`.

From RustHornBelt's point of view, `GhostPtrTokenMut` can be thought of as having two fields, `ptrs` which represents the right to free a collection of pointers, and `inner`, a `&mut GhostPtrToken`. Like a mutable reference, a `GhostPtrTokenMut`'s logical value is thought of as a current and final value. Its current value is the current value of `inner`, and its final value is the final value `inner` would have after removing all of `ptrs`. These fields would be defined as:

```
*token_mut := *(token_mut.inner)
^token_mut :=
  (^(token_mut.inner)).remove_all(token_mut.ptrs)
```

We use this idea to define the ownership predicate for `GhostPtrTokenMut` which states that: there exists a list of raw pointers (`ptrs`) and a prophetic pair (`inner`) such that `A`, `B`, `C`, and `D` hold, where:

`A` := we have the right to free all of the pointers in `ptrs`

`B` := the physical value satisfies the ownership predicate for a `&mut GhostPtrToken` with inner as its logical value

[2] We can prove relations between the final value of a mutable reference a function produces and the final value of a mutable reference a function consumes, by resolving the final value of the mutable reference the function consumed, but we cannot prove properties that are independent of the final variables of other mutable references.

C := the current value of logical value is equal to the current value of `inner`

D := the final value of the logical value of inner is equal to the final of inner with all of the pointers removed

Having defined an ownership predicate for `GhostPtrTokenMut` we can now try again to prove:

```
!(^token_mut).contains(ptr)
```

Because of the way we defined `^token_mut` above, this is equivalent to:

```
!(^(token_mut.inner))
  .remove_all(token_mut.ptrs).contains(ptr)
```

We can prove this by proving:

```
token_mut.ptrs.contains(ptr)
```

which we can prove by our choice of `token_mut.ptrs` since it is not a prophecy variable.

Using the ownership predicate we can now prove all of the specifications on a version of `take_mut` with the modified signature using `GhostPtrTokenMut`:

```
pub fn take_mut<'o, 'i>
(self: &'o mut GhostPtrTokenMut<'i, T>, ptr: *const T)
-> &'i mut T
```

When dropping a `GhostPtrTokenMut` we would like to learn that:

```
^token_mut == *token_mut
```

similar to as if it were a mutable reference. Owning `token_mut.inner` implies that we own the right to free all the memory in `*(token_mut.inner)`, that we also own the right to free all the memory in `token_mut.ptrs`, and that owning the right to free the same memory more than once is impossible. Therefore, `token_mut.inner` and `token_mut.ptrs` cannot share any elements, as that would cause us to own the same right twice, i.e., :

```
token_mut.ptrs.disjoint(*(token_mut.inner))
```

Since `token_mut.inner` is a mutable reference, dropping it allows us to learn:

```
^(token_mut.inner) == *(token_mut.inner)
```

Combining all of these facts allows us to learn the desired equality as shown below:

```
^token_mut
// by definition
== (^(token_mut.inner)).remove_all(token_mut.ptrs)
// since ^(token_mut.inner) == *(token_mut.inner)
== (*(token_mut.inner)).remove_all(token_mut.ptrs)
// since token_mut.ptrs.disjoint(*(token_mut.inner))
== (*(token_mut.inner))
// by definition
== *token_mut
```

In summary, while we were proving the soundness of `GhostPtrToken`, we found that our ownership predicate was not sufficient to prove the postcondition: `!(^^self)@.contains(ptr)` on the `take_mut` method. We then introduced the concept of a `GhostPtrTokenMut` type which behaves like a mutable reference to a `GhostPtrToken`. We were then able to prove this specification, in RustHornBelt, on a version of `take_mut` that uses this `GhostPtrTokenMut` instead of a `&mut GhostPtrToken`. The full soundness proofs for both `GhostPtrToken` and `GhostPtrTokenMut` can be found in this fork[3] of RustHornBelt.

4.6 Summary

In this chapter, we introduced the `GhostPtrToken` type, which is used for verifying the safety of data structures that use raw pointers for mutable aliasing. We then described its API methods, providing an explanation of what each method does and the meaning of their specifications. We also provided some examples of data structures that make use of `GhostPtrToken`. We then, described how this type is formalized in RustHornBelt.

4.7 Future Work

As described in Section 4.5 we introduced the `GhostPtrTokenMut` type, which is used to prove the `take_mut` specification. This type has been added to Rust-

[3] https://github.com/dewert99/rust-horn-belt/tree/thesis

HornBelt but has not been implemented as part of the GhostPtrToken API. To implement GhostPtrTokenMut the following transformation methods would need to be created:

- &'a mut GhostPtrToken<T> into a GhostPtrTokenMut<'a T>
- &'b mut GhostPtrTokenMut<'a, T> into a &'b mut GhostPtrToken<T> (deref_mut)
- &'b GhostPtrTokenMut<'a, T> into a &'b GhostPtrToken<T> (deref)

Additionally, the take_mut method would need to be updated to use this new GhostPtrTokenMut type instead of &mut GhostPtrToken.

A possible benefit of having a separate GhostPtrTokenMut type is that GhostPtrTokenMut could be implemented as a ZST, making it more memory efficient than using &mut GhostPtrToken. This idea could also be applied to &GhostPtrToken by introducing and implementing a GhostPtrTokenShr type as a ZST alternative. Note: this idea of implementing ZST equivalents to references has not been proven in RustHornBelt, but should be possible.

4.8 Related Work

As previously mentioned, Verus also supports verifying some programs that have mutable aliases. To support this, the Verus API includes the PermData<T> type which is a ZST that owns a single location in memory, and the PPtr<T> type which is a thin wrapper over a raw pointer. The API also includes a Map<K, V> type which is a ZST for storing a mapping from K's to V's (where V is a ZST). We believe most of the GhostPtrToken<T> API would be possible to implement in Verus using a Map<PPtr<T>, PermData<T>>. However, since Verus does not support functions that return mutable references, the Verus version of GhostPtrToken would not be able to have the ptr_as_mut or take_mut methods, and so it would not be possible to use it to implement the linked list's IterMut struct or the LazyAllocator.

Bibliography

[1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, Oct. 2019. doi:10.1145/3360573. → pages 13, 14, 17, 68, 69

[2] C. Community. The Coq proof assistant, 2021. URL https://coq.inria.fr/. → page 53

[3] X. Denis and J.-H. Jourdan. *Specifying and Verifying Higher-order Rust Iterators*, pages 93–110. 2023. doi:10.1007/978-3-031-30820-8_9. → page 51

[4] X. Denis, J.-H. Jourdan, and C. Marché. *Creusot: A Foundry for the Deductive Verification of Rust Programs*, pages 90–105. 2022. doi:10.1007/978-3-031-17244-1_6. → pages 13, 16, 52, 68, 69

[5] D. Ewert. Mutable reference extensionality · issue 883. GitHub, Oct. 2023. URL https://github.com/xldenis/creusot/issues/883. → page 48

[6] D. Ewert and A. Summers. GhostPtrToken: formally verifying graph-like data structures in Rust. In *Third Rust Verification Workshop*, 2023. URL https://archive.org/details/ghost-ptr-token-workshop. → page v

[7] L. Gorse, P. Müller, and A. Summers. Extended support for borrowing and lifetimes in Prusti. Master's thesis, ETH Zurich, Department of Computer Science, Oct. 2020. URL https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Lorenz_Gorse_MS_Report.pdf. → page 14

[8] J.-H. Jourdan. Bugs about ghost code · issue 436. GitHub, June 2022. URL https://github.com/xldenis/creusot/issues/436. → page 59

[9] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, Dec. 2017. doi:10.1145/3158154. → pages 27, 53

[10] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018. doi:10.1017/s0956796818000151. → page 53

[11] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7 (OOPSLA1):286–315, Apr. 2023. doi:10.1145/3586037. → pages 13, 68

[12] Z. Li, J. Wang, M. Sun, and J. C. Lui. MirChecker: Detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2021. doi:10.1145/3460120.3484541. → page 18

[13] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 2022. doi:10.1145/3519939.3523704. → pages 52, 93

[14] A. VanHattum, D. Schwartz-Narbonne, N. Chong, and A. Sampson. Verifying dynamic trait objects in Rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, May 2022. doi:10.1145/3510457.3513031. → page 18

[15] F. Wolff, A. Bílý, C. Matheja, P. Müller, and A. J. Summers. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, Oct. 2021. doi:10.1145/3485522. → pages 39, 51