ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D INFK

Eidgenössische Technische Hochschule Zürich

Master's Thesis

Verification of Closures in Rust Programs

Author:
Fabian Wolff

Supervisors:
Alexander J. Summers
Peter Müller

Programming Methodology Group
Department of Computer Science
ETH Zürich

September 30, 2020

Abstract

We consider the problem of *closure verification* in the context of Rust programs and, specifically, the Prusti project. We categorize closure occurrences in real-world code into four categories, based on a small-scale survey of Rust code on GitHub. We then provide techniques to verify many of them by refining a previously existing notion of *specification entailment* (or *fulfillment*) for ahead-of-time reasoning about all possible calls, and by introducing a novel arrow (`~~>`) notation for *a posteriori* reasoning about the effects of specific calls. We further supplement these techniques with a range of auxiliary tools, including invariants on the captured state, ghost state, and ghost arguments/results, before demonstrating how they can be encoded in Viper, an intermediate verification language, and implemented in Prusti, an automatic verifier for Rust. Our work extends the state of the art of closure verification in the existing literature, specifically in automatic verifiers based on first-order separation logic and corresponding tooling, in terms of modularity, flexibility, and ease of use.

Acknowledgments

During the course of my work on this thesis, I have had many prolonged and fruitful discussions with my main supervisor, Alexander Summers, who thereby helped to shape most of the ideas presented here. His guidance and input in that regard have been invaluable, and I am very grateful that he took the time for our weekly meetings despite his many other responsibilities and commitments as a new professor at the University of British Columbia. His lecture *Program Verification*, delivered in Zürich in the spring of 2019, provided many of the technical foundations for me being able to work on this topic in the first place. I am also grateful to Peter Müller, who supervised and coordinated this project in his group, and who first piqued my interest in program verification through his excellent *Concepts of Object-Oriented Programming* lecture. Furthermore, I would like to thank Vytautas Astrauskas and Aurel Bílý, who have provided code reviews and supported my work on the implementation in Prusti; Christoph Matheja, for his work on snapshots, and for assisting me in my first steps in the Prusti codebase; and Federico Poli, for his help in getting me started in the first few weeks of the project.

Contents

1. Introduction

Closures, sometimes also called *lambdas*,[1] are anonymous functions that can capture their lexical environment. They are a powerful and useful language feature with their origins in early functional programming languages [35] that have since found their way into many imperative and object-oriented mainstream languages, including Java, Python, C++, and Rust. They can be used for purposes such as code deduplication, as arguments to higher-order functions such as `map()` and `fold()`, as callbacks in, say, GUI frameworks, as a decorator pattern to wrap other functions, and many more.

Rust is an emerging systems programming language with a strong *ownership* type system, ensuring the absence of many memory safety issues typical of C/C++ codebases, including memory leaks, dangling pointers, and data races. In particular, the creation of mutable aliases is severely constrained in Rust—its system of ownership and borrowing ensures that no more than one mutable reference to the same object exists at any given time. This greatly facilitates verification efforts, because side-effects, including modifications to aliased memory locations, are much easier to track (or forbidden in the first place). For this reason, Astrauskas *et al.* [1] have launched the Prusti project, which aims to create an automatic verifier for Rust code that is suitable for end users without a background in verification. Given the ubiquity of closures in real-world Rust code, this thesis intends to fill the gap and provide support for closures in Prusti.

Closures can be viewed as a generalization of regular functions, being originally defined as "the λ-expression and the environment relative to which it was evaluated" [18], thus comprising a function plus an "environment", its captured state. We should therefore expect an added level of difficulty when verifying closure code, as opposed to regular functions. Closure specifications can depend on the captured state, which is invisible to the caller; indeed, closures can even *modify* their captured state and thus behave differently on every call.

Captured state is the defining specificity of closures. But effective use of closures in practice requires the use of some higher-order functional features not *strictly* related to closures, such as higher-order functions taking functions/closures as arguments or returning them, and the ability to store functions/closures in variables or struct fields. Any satisfactory approach to handling closure code in an automatic verifier will thus also benefit codebases using "only" regular functions (and passing them around, say, as function pointers).

[1] The term "lambda" originates, of course, from the lambda calculus, where an anonymous function is represented by a lambda abstraction $\lambda x.f$, where x is considered bound in f and f may have further free variables, but they are not "part of" the λ-expression, whereas a closure explicitly *captures* the free variables (its "environment"). Thus, the evaluation of a λ-expression depends on the current (or "activation", "call site") environment, whereas a closure will be evaluated in its "own" environment. Moses [24] views the analogon of a λ-expression in LISP as "a [porous] or an open covering of the function since free variables escape to the current environment", whereas by capturing the environment, we achieve "a closed or nonporous covering (hence the term 'closure' used by Landin)" (referring to Landin's 1964 paper [18]).

In this thesis, we present an approach to handling closure verification which aims to be powerful, modular, expressive, and comprehensible. We draw on the earlier work described in the next section, combined with several novel features and extensions. Our approach is tailored towards Rust, sometimes relying on its type system's guarantees, but not excessively so; thus, our approach should be applicable to and useful for verification efforts in other languages as well.

Chapter 2 will accustom the reader with the required background knowledge—introducing relevant features of the Rust language, our verification toolstack, and previous methodologies that we shall build upon—before Chapter 3 segues into a description of four categories of real-world closure occurrences, each one posing different verification challenges and motivating different strategies. The latter will be discussed in Chapter 4, detailing the specification language at the user level, whereas Chapter 5 presents the implementation of these strategies at the level of Viper, an intermediate verification language. The aptitude of our methodology for specifying and verifying a range of example programs is assessed in Chapter 6, before Chapter 7 concludes this thesis and surveys the prospects of future work on this topic.

Related work

This thesis draws many ideas from an earlier investigation into closure verification by Kassios and Müller [15], whose basic ideas are also present in Nordio *et al.*'s work [26]. Additionally, an unpublished student project report [12] contains some ideas which have been adopted, refined, and further developed here. A former Master's thesis by Weber [37] in Peter Müller's group also explored automatic verification of closures, in Python; Weber employs so-called "Call Slots", meaning specific descriptions of the states in which closure calls happen, together with manual proofs at the call-site of higher-order functions to ensure that the closure's specification actually allows it to be used in these particular locations/circumstances. This approach is burdensome and not very modular, though; also, Weber fails to present a technique for handling captured state, which by definition is essential for closures.

Svendsen *et al.* [32] discuss the very similar problem of verifying *delegates* in C#, but they resort to the use of higher-order separation logic, which is unsuitable for verification efforts based on first-order logic and SMT solvers. Several other works in the literature [10, 14, 17, 27] also rely on higher-order logic. There exist also a number of theoretical treatments of higher-order function verification [9, 13, 33], which are often unsuited for automatic verification, as they usually require a large specification overhead.

Darvas and Leino [5] as well as Leino and Müller [21] present techniques for verifying higher-order functions in imperative and object-oriented settings, but only for pure functions, whereas the approach presented in this thesis explicitly aims to support effectful closures, by themselves and as arguments to higher-order functions.

2. Background

2.1. Motivation

Closures behave somewhat like instance methods in that they have state associated with them, but unlike instance methods, a closure's captured state is hidden and inaccessible to anyone but the closure itself—at least until the closure's lifetime ends:

```
let mut count = 0;
let mut cl = || -> i32 { let r = count; count += 1; r };

assert_eq! (cl (), 0);
assert_eq! (cl (), 1);

// cl is no longer live here
assert_eq! (count, 2);
```

Closures can be stored in variables, passed around, and assigned to each other:

```
let hocl = |i: i32| { move || i };
let mut f = hocl (1);
assert_eq! (f (), 1);

let g = hocl (2);
assert! (f () != g ());

f = g;
assert_eq! (f (), g ());
```

This is especially true for boxed closures, which can be reassigned to instances of completely different closure definitions:

```
let mut f: Box<dyn FnMut (i32) -> i32> = Box::new (|i| 42);

let mut x = 0;
f = Box::new (move |i| {let r = x; x = i; r });
```

Moreover, closures are often used in the context of higher-order functions, i.e. functions that take other functions/closures as arguments, or return them:

```
let nums = vec! [1, 2, 3];
let doubled = nums.iter ().map(|i| i * 2).collect::<Vec<_>> ();
assert_eq! (doubled, vec! [2, 4, 6]);
```

A modular approach to verification demands the higher-order function to be verified independently from its argument closures. Therefore, we must be able to verify closure calls without knowing *which* closure gets called (and, by extension, what precise specification it has, what kind of captured state, *etc.*).

Yet the higher-order function has to know whether it can call the closure, may expect some properties about its behavior, and needs a way to describe the effects/result of the closure call:

```
fn foo (mut f: impl FnMut (i32) -> i32) -> i32 {
    // Does f's precondition hold?
    let x = f (42);

    // f should not return 0
    assert_ne! (x, 0);

    // What should foo's postcondition be?
    x
}
```

Therefore, there are two angles from which we need to reason about closure calls: Before we can call the closure, we need to know that its precondition holds, for a given set of arguments and the current value of the captured state.[1] Moreover, a higher-order function might want to place certain restrictions on the behavior of the closure; say it wants to divide by the closure's return value, then the closure's postcondition should imply that its result is non-zero. Note how these kinds of considerations happen *before* the closure was called; thus, we will refer to them as *a priori* reasoning.

The second angle is about describing the *effects* of specific closure calls. This is easier for regular function calls, because we know which function we are calling, and thereby we also know that function's concrete specification.[2] For closures, however, this information might not be available statically and/or modularly: A higher-order function can require certain restrictions on its argument closure's behavior, as described above, but these restrictions must be weak enough to be fulfilled by all closures we might want to pass into this function. Thus, a higher-order function usually does not have any concrete knowledge about its argument closure's behavior and must express its own behavior abstractly, or parametrically. For this, we need a way to talk about the effects of closure calls abstractly, *after* they have happened, i.e. from an *a posteriori* point of view.

2.2. Languages and Tooling

2.2.1. Closures in Rust

The Rust programming language prevents certain classes of errors related to memory management statically, through the use of an advanced *ownership* type system, in which every value always has exactly one *owner*, a variable (and, by extension, a thread). Sharing of data (between, say, different functions or

[1] Implicating the captured state in the precondition may sound unintuitive, because the closure's caller does not have access to it; however, one can construct examples where, say, a closure should be called at most n times, and this could be implemented by a counter in the captured state that is exposed to the caller (via *views*, as we will see in Section 4.1.1) and mentioned in the precondition.

[2] This is not true for calling dynamically dispatched methods on trait objects, because the concrete subtype might not be known, but in that case, we still know the trait method's specification.

threads) can happen via *moves* (ownership transfers), copies, and *borrowing*, which refers to the creation of non-owning references to an object, either immutable or mutable. The type system guarantees that only an owned value may be moved or deallocated, and only if there are no live references to it. Furthermore, at most one mutable reference may exist at a time, and only if there are no other currently-usable references to the same object, to prevent race conditions and provide *framing* information: A referenced object is never modified unexpectedly by another actor (such as a called function or another thread).[3]

What is special about Rust closures, compared to the same feature in other languages, is that the aforementioned rules for value sharing apply; in particular, Rust closures capture their environment by *borrowing* values immutably or mutably or by *moving* or copying values into them. In the following example, `cl` borrows `x` immutably, `y` mutably, and `z` is moved into `cl`:

```
let mut x = "x".to_owned ();
let mut y = "y".to_owned ();
let mut z = "z".to_owned ();
let x_borrow = &x; // we can have overlapping immutable borrows

// || introduces a closure; arguments go between the bars:
let cl = || {
    let x_clone = x.clone (); // this only reads from x
    y.push_str (&x_clone); // modify y
    std::mem::drop (z); // this call requires ownership of z
};

cl ();
println! ("{} {}", x_borrow, y); // we can't use z anymore here
```

A Rust closure is modeled internally similar to a `struct` with one field for a function pointer to the code and additional fields for every captured value of appropriate types. Because of this, every closure definition is given a unique and anonymous type by the compiler, which is why closure variables cannot be reassigned even to syntactically equal closures:

```
let mut cl = |i: i32| -> i32 { i };
cl = |i: i32| -> i32 { i }; // error: mismatched types
```

Note, however, that multiple instantiations ("instances") of the *same* definition are all going to have the same type. This can happen whenever the definition is, say, inside of a loop or a function/closure, and thus can be passed more than once by the control flow. Instances of the same definition, therefore, can be assigned to each other:

```
let ho_cl = |i: i32| { move || -> i32 { i } };
let mut a = ho_cl (1);
let mut b = ho_cl (2);
a = b;
```

Every closure in Rust implements one or more of the three common supertypes (*traits*) for closures in Rust:

[3] An exception to this rule is the notion of *interior mutability* used by some libraries; this requires the use of `unsafe` code to circumvent some of the restrictions.

1. The `Fn` trait, which is implemented by all closures that can be called via immutable references (i.e., that do not modify or move their captured state). Closures of this type can be called arbitrarily often.

2. The `FnMut` trait, which is implemented by all closures that can be called via mutable references (and thus *may* modify, but not move, their captured state). Closures of this type can also be called arbitrarily often.

3. The `FnOnce` trait, which is implemented by all closures. Closures of this type can be called at least once; closures that move out their captured state can be called *at most* once and thus implement only this trait.

In particular, closures that *don't* capture any state, even immutably, can be coerced into simple function pointers:

```
let f: fn() -> i32 = || 42;
```

At the same time, simple functions can always be wrapped into `Fn`-trait objects, because they don't have any captured state and thus can't modify or move it:

```
fn g_function () -> i32 { 42 }
let g: Box<dyn Fn() -> i32> = Box::new (g_function);
```

For further reading on Rust, both in general and relating to closures specifically, please refer to the Rust documentation; in particular, "the book" by Klabnik and Nichols [16], the community-maintained "Rust by Example" [29], and, for examples of closure occurrences as well as documentation about standardized higher-order functions that work with closures, the Rust standard library manual [34].

2.2.2. Viper

Viper [25] is a verification infrastructure, meaning it is a language for expressing programs and properties about such programs, together with an implementation that verifies these properties. Unlike previously-existing verification infrastructures such as Boogie [19] and Why [8], Viper was developed with an eye toward separation logic and permission-based reasoning [28], which it natively supports.

Viper is considered an intermediate language, meaning it is intended to be used as a basis for front-end verification tools (Prusti, described in Section 2.2.3 below, being one example of the latter). For this reason, Viper supports different styles of reasoning (e.g., recursive predicates as well as quantified permissions [25, Section 3]) to provide flexibility for the front-end tools, as well as having many features relevant for encoding imperative and object-oriented front-end languages built-in, such as heap references, loops and other control structures, heap-dependent (pure) functions and (impure) methods, custom types (via *domains*), and more.

The core of separation logic is a notion of *permissions* to fields of heap objects. Fields are defined globally in Viper:

```
field a: Int
field b: Bool
```

Fields can be accessed via references:

```
var r: Ref
var i: Int := r.a // fails
```

This snippet would fail to verify, because we don't hold any permissions to `r.a`. Permissions are represented by the `acc` keyword; `acc(r.a)` means full (exclusive) permission to field `a` of object `r`, and `acc(r.a, 1/2)` means a *fractional* permission to the same field: A full, exclusive permission is required for write access, and any non-zero permission is sufficient for read access.

Permissions can be added to the current state via *inhales*:

```
var r: Ref // fresh variable, we have no permissions
inhale acc(r.a, 1/2) // adds 1/2 permission to r.a
inhale acc(r.a, 1/2) // we now have full permission to r.a

inhale acc(r.a, 1/2) // permission exceeds 1 now
```

Holding more than a full permission (i.e., a permission greater than 1) to a field in a state amounts to the same effect as writing `assume false`.

`exhale` has the opposite effect of *removing* permissions from the state:

```
var r: Ref
inhale acc(r.a) // we now have full permission to r.a
exhale acc(r.a, 1/3) // we still hold 2/3 permission
exhale acc(r.a) // fails, can't have negative permissions
```

Exhaling permissions that are not currently held is a verification error, equivalent to an `assert false`, such as in the last line of the snippet above, where we try to exhale a full permission while only holding `2/3`.

The value of permissions lies in their ability to provide us with *framing* information: Because permissions to a field of a given object need always add up to one, we know that as long as we hold a positive amount of fractional permission, nobody else can have full permission, and thus nobody can modify the field unexpectedly. If, however, we give up all of our permissions, then we lose all knowledge about the field's content, even after regaining permission:

```
var r: Ref
inhale acc(r.a, 1/2) && r.a == 42
assert r.a == 42 // succeeds

exhale acc(r.a, 1/2)
assert r.a == 42 // fails, insufficient permission

inhale acc(r.a)
assert r.a == 42 // fails, value could have been changed
```

Permission transfers often happen at method calls:

```
method inc_a(r: Ref)
  requires acc(r.a)
  ensures acc(r.a) && r.a > old(r.a)
{
    r.a = r.a + 1;
```

```
}

var r: Ref
inhale acc(r.a) && r.a == 0
inc_a (r)
assert r.a > 0
```

A method call amounts to exhaling the precondition, potentially losing some or all permissions to certain fields and thus possibly some framing information, and inhaling the postcondition, potentially (re)gaining permissions and some knowledge about the values of certain fields.

Further reading on Viper may be found in the main publication by Müller *et al.* [25], in the tutorial [36], and in several more specialized papers on the implementation of Viper, such as Schwerhoff and Summer's treatise [30] on automated handling of magic wands, and on applications of Viper to the verification of real-world problems, such as an encoding of weak-memory programs by Summers and Müller [31].

2.2.3. Prusti

The Prusti project exploits the safety guarantees of Rust's type system for automatic verification purposes. The input Rust program is encoded in Viper; and in particular, type information from the Rust compiler is used to construct a so-called "core proof", which supplies all the necessary separation logic annotations to allow for automatic verification in Viper to go through. This re-verifies the memory safety properties already given by the type system, but it also allows us to give further specifications, concerning functional behavior, or safety properties, such as the absence of crashes and integer overflows. These stronger specifications will be integrated into the core proof and checked modularly and automatically. [1]

If no functional specifications are given, and checks for integer overflows and crashes/assertion failures are disabled, no annotations need be supplied at all (because the memory safety properties verified in this case won't go beyond what is already given by the type system, as described above).

Prusti is able to verify that assert statements such as the following will not fail at runtime:

```
let x = y * y;
assert! (x >= 0);
```

Additionally, we can supply functional specifications in the form of pre- and postconditions, which need to be proved where the function is defined:

```
#[requires(i >= 2)]
#[ensures(result > 10)]
fn foo (i: i32) -> i32 {
    i * 8
}
```

This knowledge about `foo`'s behavior can (and must, for the precondition) now be used at every call-site of `foo`:

```
if y < 2 { y = 2; }
let z = foo (y);
assert! (z / 2 >= 5);
```

The type system gives us framing information: As long as we hold a currently-usable[4] reference (corresponding to a non-zero amount of permissions) to an object, nobody else can modify it; for write access, we need a mutable reference (a full/exclusive permission); passing around a mutable reference (transferring permissions) allows others to modify the object and thus loses framing information.

Since reasoning about Rust code in this way does not require the intimate knowledge of permissions and separation logic necessary for the effective use of lower-level tools like Viper, the Prusti project hopes "that it lowers the barrier to applying verification" [1], both from a perspective of time/effort (since the construction of the core proof is automatic; so users can focus on functional and safety properties, rather than worrying about, say, the correct amounts of permissions needing to be passed around, *etc.*) and one of knowledge/experience (not requiring a background in verification might enable and encourage more users to adopt Prusti for their own projects).

For further reading on Prusti, refer to the main publication by Astrauskas *et al.* [1].

2.3. Previous Methodology

2.3.1. Specification Functions

Kassios and Müller [15] present a verification methodology to reason about closures based on so-called *specification functions*. When a closure is stored in a variable or passed as an argument, it is, in general, unknown where the closure was defined.[5] In addition, the same closure *definition* can lead to many different *instances*, as in the following example (adapted from the Kassios/Müller paper, although Kassios and Müller do not work with Rust code):

```
let counter = |x: u32| {
    let mut count: u32 = x;
    let mut inc = move || { let r = count; count += 1; r };
    inc ();
    inc
};
```

This necessitates a mechanism for "looking up" closure specifications. For this purpose, Kassios and Müller define their *specification functions*, one for each closure signature (as the concrete closure could be any one with a matching signature). Let $CI_{(T_1,T_2,\ldots)\to R}$ denote a common supertype for closures

[4] Rust has "non-lexical lifetimes", meaning the borrow checker can deduce a reference to be dead and treat it as such even while it is still in scope.

[5] In Rust, every closure definition is assigned a unique type during compilation, so we could *in principle* keep track of (say) all instantiations of a generic higher-order function for concrete closure types. This is non-modular, though, because the higher-order function's specification would then have to be re-checked for every instantiation. Furthermore, this wouldn't work for dynamic references (or boxes) to one of the `Fn*` traits, because they point to a *dynamic* subtype, i.e. the concrete closure type (and therefore, its definition) won't be known until runtime in general.

with the signature $(T_1, T_2, \dots) \to R$. Then the pre- and postcondition specification functions will have the following signature:[6]

$$pre_{(T_1, T_2, \dots) \to \mathrm{R}} : \left(CI_{(T_1, T_2, \dots) \to \mathrm{R}},\ Heap,\ T_1,\ T_2,\ \dots\right) \to Bool$$
$$post_{(T_1, T_2, \dots) \to \mathrm{R}} : \left(CI_{(T_1, T_2, \dots) \to \mathrm{R}},\ Heap,\ Heap,\ T_1,\ T_2,\ \dots, R\right) \to Bool$$

Note how the postcondition function receives *two* heaps, because it needs to relate the pre- to the poststate. The arguments may be heap-dependent, i.e. if, say, T_1 is a pointer/reference type, then dereferencing it may yield different results in the pre-/poststate heap.

We can use the specification functions to describe the behavior of higher-order functions, including `counter` from above:

```
let counter =
  // ensures: ∀ h : Heap :: pre (result, h)
  |x: u32| { ... };
```

to specify that `counter`'s result has precondition `true`. The postcondition is expressed through similar means, although for this example, we need a further specification function abs_T that "abstracts" (and thereby, more importantly, exposes) parts of the captured state (the subscript type in abs_T is a technical necessity, as the type of the abstracted captured state is not apparent from the closure's signature; Kassios and Müller extend the closure signature to make this explicit,[7] but we will omit it here, as it is not important for the remainder of this section):

```
// ensures: abs(result) = x + 1
// ensures: ∀ oldh, h : Heap, r : Int :: post(result, oldh, h, r) ⇒
//          (r = abs(result, oldh) ∧ abs(result, h) = abs(result, oldh) + 1)
```

Here, *abs* with just one argument denotes a special overload that is to be evaluated in the current heap (i.e., `counter`'s poststate). x (in $abs(result) = x + 1$) refers to `counter`'s argument.

Now, to encode a function call $f(t_1, t_2, \dots)$, we can write (somewhat simplified from Kassios and Müller):

assert $pre_{(T_1, T_2, \dots) \to R}$ (H[f], H)
oldHeap := H
havoc H, r
assume $post_{(T_1, T_2, \dots) \to R}$ (oldHeap[f], oldHeap, H, r)

for fresh r and square brackets denoting heap lookups (writing H[f] is necessary because f, the concrete closure instance, is not known statically; it could be any closure with a matching signature, stored behind a reference, and thus we need to look it up on the heap).

[6] As a technical necessity, Kassios and Müller additionally pass an "allocation table" to the specification functions, which they need for encoding allocations/deallocations inside the closure body. We omit this here for simplicity and because it is not relevant to our approach: Rust's type system gives us all the knowledge we need about (de-)allocations inside closures/functions.

[7] This is actually a modularity problem, at least without adequate subtyping rules in place, because closures with different types of captured state now no longer have the same signature; a higher-order function will have to decide which kind of captured state its argument closures may have, *etc.*

To encode calls to higher-order functions, no further action is necessary: The higher-order function's specification will probably mention *pre* and/or *post* with regards to its argument or result, but the SMT solver can use whatever knowledge it currently has about these functions (gained either by knowing the concrete closure instance and its specification, or via the postcondition of some other higher-order function, or through a manual assume statement, or …) to verify a call to a higher-order function in the same way that a regular function call is handled. For further details on this and other aspects of specification functions, please refer to the work of Kassios and Müller.

This small "counter" example demonstrates the flexibility of specification functions, but also several of their weaknesses, especially as a part of the user-level specification language: Specification functions are not always particularly easy to understand and use for users without a verification background—a target group of the Prusti project—, and even for experienced users, having to manually quantify over heaps and such is a bit of a nuisance and not extremely readable. In fact, implementing this technique requires heaps to be first-class objects, to be passed around and quantified over, which is neither possible in Viper nor compatible with separation logic in general.

2.3.2. Specification Entailments

Hader [12] takes a different approach: At the core of his strategy lie specification *entailments*. To reason about specifications in assertions, he defines a custom syntax:

$$|\, t_1 : T_1,\ t_2 : T_2,\ \ldots \,|\ \ \{\ \textbf{requires:}\ P\,(t_1, t_2, \ldots)\,,\ \ \textbf{ensures:}\ Q\,(t_1, t_2, \ldots)\ \ \}$$

where $t_1, t_2, \ldots$ are binders for the closure arguments (Hader mandates their types to be given explicitly, but these could also be inferred from the context where possible), P is an expression for the precondition and Q is an expression for the postcondition, which may additionally refer to the **result** as well as wrap sub-expressions into **old**() to refer to the prestate. Either P or Q, or both, may be omitted and default to "true". The meaning of such a specification is an implicit[8] quantification not only over the arguments, but also over states: In *any* state, for any arguments $t_1, t_2, \ldots$, if P holds, we may call the function and assume Q in the poststate.

Now, assuming we have such specifications spec1 and spec2, we can use the entailment operator "⊨":

$$\begin{aligned} \text{spec1}: &\ \ |\,\text{params1}\,|\ \ \{\ \textbf{requires:}\ \text{pre1},\ \ \textbf{ensures:}\ \text{post1}\ \} \\ \text{spec2}: &\ \ |\,\text{params2}\,|\ \ \{\ \textbf{requires:}\ \text{pre2},\ \ \textbf{ensures:}\ \text{post2}\ \} \end{aligned}$$

$$\text{spec1} \models \text{spec2}$$

This is a logical assertion that can be either true or false, expressing that

1. the signatures (params1 and params2) of spec1 and spec2 match;
2. spec1 has a weaker precondition than spec2 ($\text{pre2} \Rightarrow \text{pre1}$); and
3. spec1 has a stronger postcondition than spec2 ($\textbf{old}(\text{pre2}) \Rightarrow (\text{post1} \Rightarrow \text{post2})$).

[8] This is the analogon to Kassios and Müller's explicit quantification over heaps.

Intuitively, this means that wherever a closure with specification spec2 is expected, we can pass in a closure with specification spec1; this closure *also* fulfills spec2, hence the term "entailment" (knowing that spec1 holds entails knowing that spec2 also holds).

Note that these rules are equivalent[9] to the behavioral subtyping rules for method overloads in object-oriented programming languages, pioneered by Liskov and Wing [23] and subsequently refined by Dhara and Leavens [6]. Thus, we could say that spec1 represents a behavioral subtype of spec2.[10]

For practical purposes, where the precise (concrete) specification of a closure f is often not known and we want to check whether f fulfills a given specification—for instance, f could be an argument to a higher-order function—, we can write

$$f \models |\text{params}| \ \{ \ \textbf{requires:} \ \text{pre}, \ \textbf{ensures:} \ \text{post} \ \}$$

to express that the concrete (potentially unknown) specification of f entails the given specification (call it spec3). Assuming this is part of the precondition of a higher-order function, the caller, who might know the concrete closure instance f (and, therefore, its specification), would have to check whether the entailment holds. But even if f's concrete specification is *not* known, we may know that a different entailment holds (e.g. $f \models \text{spec1}$) and use that to prove the above entailment ($\text{spec1} \models \text{spec3}$ in this example).

In order to encode these entailments without pushing them down to the SMT solver—in other words, to know even which entailments to encode (such as $\text{spec1} \models \text{spec3}$ in the example above)—, we need to keep track of the entailments. For this reason, Hader mandates that specifications may occur only in conjunctions and as consequents (but not antecedents) in implications, and not under quantifiers, disjunctions, and negations. That way, we know which entailments hold at each program point (perhaps conditionally, depending on the implications) and can use this information for encoding further entailments and function calls. For instance, assume we don't know the concrete specification of f, but we do know $f \models \text{spec1}$. Then, to encode a call to f, we must assert pre1 and assume post1 (with proper variable substitutions for the arguments to the call, *etc.*), because we can't assert/assume the actual, unknown specification of f.

This presents certain challenges and limitations. In particular, assume that, at some point in our program, we know both $f \models \text{spec1}$ and $f \models \text{spec2}$; then, to check whether $f \models \text{spec3}$, we need special handling to encode that spec1 and spec2 *together* entail spec3, such as in the example given in Figure 2.1. It is also not clear how to extend this approach to support quantifiers, at least without reimplementing triggering/instantiation logic from the SMT solver.

Another issue is the fact that there is no straightforward way to specify, say, the classic *map* higher-order function (given for integers here, for simplicity):

```
// requires: f |= |i: i32| { requires: true, ensures: ??? }
fn map (self, f: impl Fn (i32) -> i32) -> ...
```

What should we substitute for `???` above? We can't give a more concrete postcondition than `true`,

[9] Some object-oriented languages may additionally allow the argument types to be contravariant and the return type to be covariant; here, we assume matching signatures, because Rust does not support subtyping of function/closure types with different signatures.

[10] Hader defines entailment the other way round, but then uses it the way it's defined here; we believe this to be an error in his work.

$$\begin{aligned}
\text{spec1}: &\ |i : \text{Int}|\ \{\ \textbf{requires:}\ i \geq 0,\ \ \textbf{ensures:}\ \textbf{result} > 0\ \} \\
\text{spec2}: &\ |i : \text{Int}|\ \{\ \textbf{requires:}\ i \leq 0,\ \ \textbf{ensures:}\ \textbf{result} > 0\ \} \\
\text{spec3}: &\ |i : \text{Int}|\ \{\ \textbf{requires:}\ \textbf{true},\ \ \textbf{ensures:}\ \textbf{result} > 0\ \}
\end{aligned}$$

Figure 2.1.: Example of two specifications together entailing a third.

because `map` should work for all argument closures; but placing `true` in the specification above will give us a very weak postcondition of `map`.

Hader proposes passing a ghost argument into `map`:

```
// ghost_args: P: Pred (i32, i32)
// requires: f |= |i: i32| { requires: true, ensures: P (i, result) }
// ensures: forall idx :: 0 <= idx && idx < self.length
//     ==> P (self [idx], result [idx])
fn map (self, f: impl Fn (i32) -> i32) -> ...
```

Predicates (used here in the sense of boolean pure functions) as ghost arguments are indeed a useful and powerful feature, but they are not a satisfactory solution to this problem: First, `map` doesn't really *care* about `f`'s postcondition, so it would be desirable to have a way to "decouple" `f`'s behavior from that of `map`, and second, the caller of `map` typically already knows `f`'s postcondition, so it is a nuisance to have to specify `P` explicitly (although this problem could be solved by trying to *infer* a suitable `P` automatically at the call-site, which, although not discussed in Hader's work, might be doable in principle).

Some other problems remain, too. Most notably, to allow the caller of a higher-order function to reason about modifications to the closure's captured state, we have to expose the captured state to the higher-order function, so that it can reason about it in its specification. This is inflexible and non-modular, however, because the higher-order function shouldn't, and in fact can't, know which kinds of captured state its argument closures may have, how they modify it, *etc.* We need a way to opaquely reason about the closure's behavior and captured state, so that they don't have to be exposed to higher-order functions, but the caller still knows what the higher-order function did with the closure.

3. Classification

By studying closure occurrences in real-world Rust code on GitHub (informally; that is, without a representative, statistical analysis), we have identified four different *classes* of closure uses, each one describing a different (although potentially overlapping) use case, and, in particular, each one entailing different verification challenges and thus motivating different verification techniques, which will be discussed in Chapter 4. Afterwards, as part of the evaluation, Section 6.1 will discuss fully worked out example specifications for instances of each of the classes given in this chapter.

In this chapter, we will sometimes refer to standard library functions as examples for the various categories; for more information about these functions, refer to the standard library documentation [34].

3.1. Higher-Order Functions over Collections

This category encompasses the traditional, "classic" higher-order functions like `map()` and `fold()`. Their defining characteristic is that they take an argument closure and call it an unbounded amount of times while iterating over a collection of elements, each time passing different arguments to the closure. The precise order of the calls typically isn't extremely relevant from a verification perspective, as most `map()`s and `filter()`s work in a point-wise fashion, and even many `fold()`s are associative and commutative (for instance, using `fold()` to sum up integers). Furthermore, while argument closures to these higher-order functions can have side-effects, it is often unidiomatic to depend on the *order* of these side-effects.

The challenge here is to write specifications that are

- *flexible*, from the caller's perspective: We want to be able to pass any function into `map()` or `fold()`, regardless of their concrete behavior. At the same time, we want to be able to integrate knowledge about the collection we're working on—for instance, if we know that a `Vec<i32>` contains only positive values, we'd like to be able to pass a closure into `map()` whose precondition states that its argument must be nonnegative.

- *modular*—the higher-order function should not need to know anything about the closure's behavior (or the collection's contents, for that matter); rather, it should have a "parametric" specification, allowing the caller to use her knowledge of the closure and the collection to reason about the higher-order function's precise effects at the call-site.

- *powerful*—of course, we want to capture the higher-order function's behavior as well as possible. For instance, for a call to `fold()`, it is not enough to prove that, say, its result has a certain property that is implied directly by the closure's postcondition. Rather, we want to capture the actual functional behavior of `fold()`:[1]

[1] At least to the extent permitted by our underlying verification infrastructure; for instance, at the time of writing, Viper

```
let nums = vec! [1, 2, 3, 4];
let cl =
    // requires: a >= 0 && c >= 0
    // ensures: result == a + c
    // ensures: result >= 0
    |a, c| { a + c };
let a = nums.iter ().fold (0, cl);
assert! (a >= 0); // easy
assert_eq! (a, 10); // hard(er)
```

To verify programs from this category, we will need a way to specify for which arguments (namely, the ones in the collection we are working on) the higher-order function must be able to call the closure. Additionally, we need a way to describe how the original collection has been changed (in case we're iterating over mutable references), and what the result (a collection for `map()` and `filter()`, a scalar for `fold()`) looks like.

3.2. Higher-Order Functions with Fixed Behavior

"Fixed behavior" in this case means that it is known statically which closures will be called,[2] how often, with which arguments, in what order, and in which states. This is in contrast with the functions from the previous section, which call their argument closures an unbounded amount of times and where the order of the calls is often irrelevant and/or hard to capture.

The challenge here is to write a specification which is *as precise as possible*, because we know exactly what calls will happen. But at the same time, the specification should remain modular, so that a client can reason about the effects on, say, the argument closure's captured state, without needing to expose it to the higher-order function. This would be non-modular and inflexible, because the higher-order function's specification would have to be adapted every time for the precise kind of captured state.

A classic example that would fit into this category is function composition:

```
fn compose<A, B, C> (mut f: impl FnMut (A) -> B,
                     mut g: impl FnMut (B) -> C) -> impl FnMut (A) -> C
{
    move |a: A| g (f (a))
}
```

Furthermore, in Rust, every higher-order function that takes an `FnOnce` closure (and thus calls it at most once, typically in a specific state under specific circumstances) falls into this category; the ample set of examples includes `Option::map()`, `Result::map_err()`, `hash_map::Entry::or_insert_with`, and more.

---

does not support set comprehensions, which would be necessary for even expressing things like a sum over a collection of integers formally. However, these are orthogonal problems; the techniques presented in this thesis should work out of the box with a potential future implementation of set comprehensions in Viper.

[2] The closure definition that gets called may not be known; what is meant here is that it is known, say, that some argument closure `f` is called exactly once.

For this category, we need precise tools for describing exactly in which state(s) the closure's precondition must hold, as well as for describing and relating the precise state(s) in which the call(s) happened and which effects they manifested.

3.3. `sort_by()`

`sort_by()` sorts a vector according to some ordering relation, given by an argument closure. `sort_by()` warrants a category of its own, because its specification needs to express non-trivial properties about its argument closure as well as its result:

- The comparator closure needs to be reflexive, antisymmetric, and transitive. The latter two properties, in particular, are not even readily expressible in normal specifications, because they relate *pairs* and *triples* of calls together (e.g. "for all pairs of calls $f(a, b)$ and $f(b, a)$, if both calls return `true`, then $a = b$").
- The order of elements in the result not only matters but needs to be expressed in terms of the argument closure's behavior: "The result is sorted according to the ordering relation implemented by the comparator function.".

Other examples that go in a similar direction are the `Vec::dedup_by()` and `slice::partition_dedup_by()` functions, which deduplicate vectors/slices according to the equality relation given by their argument closures. These comparator closures thus need to be reflexive, symmetric, and transitive.

3.4. Boxed Closures, Dynamic References to Closures, and Function Pointers

This category is less about how the closure gets *used*, and more about how it is *stored*. Every closure gets a unique type in Rust, so it is impossible to reassign closure variables except to different instances from the same definition, as described in Section 2.2.1. This provides framing information in the sense that a closure (meaning the executable part of it) stored in a variable can't ever be changed, and thus its behavior also cannot change, only the value of its captured state (which may, of course, influence the behavior, but nonetheless it is always the same code that is being executed). The same is not true for closures stored in boxes or accessed through dynamic references and function pointers:

```
// Function pointer:
let mut cl_fp: fn (i32) -> i32 = |i: i32| -> i32 { i };
cl_fp = |i: i32| -> i32 { i + 1 };

// Boxed closure ("dyn" because the concrete type is a dynamic subtype):
let mut cl_box: Box<dyn Fn (_) -> _> = Box::new (|i: i32| -> i32 { i });
cl_box = Box::new (|i: i32| -> i32 { i + 1 });

// Dynamic reference:
let mut cl_ref: &dyn Fn (i32) -> i32 = &(|i: i32| -> i32 { i });
cl_ref = &(|i: i32| -> i32 { i + 1 });
```

This is a useful feature for storing a closure in a field of a struct, for instance; imagine a callback function of sorts that can be accessed and changed via getter and setter functions as an example. However, this feature also poses additional challenges from a verification perspective: Knowing that, say, `*cl_ref` was called with a certain argument no longer tells us anything about the result if we don't *also* know which closure `cl_ref` referred to at the time of the call (in fact, we wouldn't even be allowed to call the closure because we wouldn't know whether the precondition holds).

4. Methodology

In this chapter, we will present techniques for the basic verification of (first-order) closures in Section 4.1, before examining higher-order function (and, in principle, closure) verification from two different angles: First, the ahead-of-time, *a priori* angle, which allows us to reason about future closure calls in unknown states and with unknown arguments. The questions we need to answer here include: Are we allowed to call a certain closure? If so, in which state(s)? What behavior do we expect from the closure? These challenges are solved by *specification entailments*, discussed in Section 4.2.

Second, and dually/complementarily to the previous point, there is an *a posteriori* perspective, to reason about the effects that have manifested in concrete calls, with known arguments and relating specific states. The challenge here is to describe a higher-order function's behavior, which depends on the behavior and side-effects of its argument closures, *modularly* (or *parametrically*); that is, without relying on any specific behavior of the argument closures in the higher-order function's specification. In other words, we want to *decouple* the higher-order function's specification from that of its argument closures, which is achieved by the *arrow notation* presented in Section 4.3.

Finally, we will present *ghost arguments* and *results*, particularly ghost argument/result *functions*, which will prove especially useful for the verification of certain higher-order functions such as `fold()` in Section 4.4.

4.1. Closure Specifications

Before we can talk about the specifications and effects of higher-order functions, we need a way to express the behavior of closures themselves, i.e. a way to attach specifications to closure definitions.[1] Just as regular functions, closures will have pre- and postconditions:

```
let cl =
    // requires: i >= 0
    // ensures: result % 2 == 0
    |i: i32| -> i32 { ... };
```

In addition, though, closure specifications may depend on captured state. Just as a function's specification must be proved in all states, for all arguments, so does a closure specification have to be proved in all states, for all arguments, and for all values of the captured state. This is a sound overapproximation—but not all values of the captured state are actually reachable, depending on the closure's behavior. In Section 4.1.3, therefore, we will examine ways to restrict this implicit quantification over states to "relevant" states, using invariants. We will also look at how to reason about the captured state from the outside using "views" (Section 4.1.1), how to include knowledge about the values in the enclosing scope in that reasoning (Section 4.1.2), and a possible extension of the captured state with ghost state for additional flexibility (Section 4.1.4).

[1] The foundations for this section are from Hader's work [12].

4.1.1. Exposing the Captured State through Views

The captured state can be made visible to the outside (for specification purposes only) as *views*, which are expressions composed of constants and captured state; in adherence to the principle of information hiding, the specification may only talk about views (and not the invisible, unexposed rest of the captured state). Views must not have side-effects; in particular, they cannot be used to mutate the captured state, and they may only have copy (i.e., implementing the `Copy` trait) or reference types—it is not allowed to move out values from the captured state through views. Here is a simple example:

```
let mut dist: i32 = 0;
let mut walk =
    // views: km: i32 = dist / 1000
    // requires: meters >= 0
    // ensures: old(km) <= km       // OK
    // ensures: old(dist) <= dist   // Error
    |meters: i32| -> i32 { dist += meters; dist };
```

Views are conceptually attached to the closure type, so whenever a closure of a certain type exists, the views associated with it are also visible. In particular, though, a higher-order function receiving a closure argument of unknown type (say, `T: impl Fn...`) won't be able to access its views (which makes sense, because it should work for *any* concrete type, which could have any set of views).

Unlike Hader, we also allow for views to take arguments; for instance:[2]

```
let mut nums = vec! [1, 2, 3];
let cl =
    // view: el: (idx: usize) -> Option<i32> = nums.get(idx)
    |i: usize, v: i32| {
        let r = nums[i];
        nums[i] = v;
        r
    };
```

4.1.2. Reasoning about the Enclosing Scope

Mutable captured state must be havoced when proving specifications, as it may be modified at will by the closure:

```
let mut j: i32 = 42;
let mut cl =
    // does not ensure anything (except true)
    || { j = rand::random (); j };
```

On the other hand, we might expect information about immutable captured state to be preserved when proving the specification:

```
let i: i32 = 42;
let cl =
```

[2] The view returns an `Option`, to account for the possibility of `idx` being out of range for `nums`. Views are ghost code and thus must not have side effects; in particular, they may not cause panics.

```
    // ensures: result == 42
    || { return i; };
```

Unfortunately, this won't work, because the Rust compiler will encode a closure definition as a regular function, taken out of the context wherein it was defined, and thus the information about the enclosing scope will be lost. In particular, for the example above, when proving the closure body, we won't know modularly (with the closure definition taken out of context) that `i == 42`, and thus we won't be able to prove the specification. Instead, we can prove the following:

```
let i = 42;
let cl =
    // view: captured_i: i32 = i
    // ensures: result == captured_i
    || i;
assert_eq! (42, cl ());
```

This may be slightly less explicit and intuitive, but it essentially does not lose any information, because at the instantiation-site of the closure (where the closure instance is created, i.e. the assignment to `cl` above), we can combine the closure specification with our knowledge of the captured state (`i` in this case). In addition to the `assert_eq!()`, we could also prove `cl |= || { ensures: result == 42 }` (using a specification entailment from Section 4.2), because that, too, would be evaluated at the instantiation-site.

4.1.3. Invariants

Single-State Invariants

Specifications talk about behavior that a closure must adhere to in all states, for all arguments. But sometimes, when working with mutable captured state, this interpretation is too restrictive. Consider the following example:

```
let mut count: i32 = 0;
let mut inc =
    // view: count: i32 = count
    // ensures: result >= 0       // fails
    move || { let r = count; count += 1; r };
```

We cannot currently prove this postcondition, because `count` is mutable captured state, and so we need to havoc it when proving the specification, as an overapproximation to guarantee correctness (at the expense of completeness). This suggests that there should be a way of attaching *invariants* to closures, which must be established by the initialization and preserved by every call (thanks to the guarantees made by Rust's type system, we know that nobody else can have mutable access to the captured state, so calls (and assignments, see below) are the only way in which the captured state can be modified):

```
let mut count: i32 = 0;
let mut inc =
    // view: count: i32 = count
    // invariant: count >= 0
    // ensures: result >= 0
    move || { let r = count; count += 1; r };
```

This allows us to prove the postcondition of `inc`. The invariant will be checked "on entry" (i.e., when the closure instance is created) and then implicitly added to the pre- and postcondition, to ensure preservation and to allow it to be used when proving the body. Again, the invariants may only talk about views, i.e. the exposed part of the captured state, and not "hidden" captured variables.

History Invariants

Single-state invariants are useful, but not yet sufficient to prove examples like the following:

```
let a = inc ();
// arbitrary code in between (but no assignments to inc)
let b = inc ();
assert! (a < b);
```

Here, a simple invariant is not enough: We need a way to relate different "versions" of the captured state across potentially many (and a statically-unknown number of) calls. Luckily, the captured state is not modified in a completely haphazard manner, but in an orderly fashion, and we can capture this using a *history invariant*. History invariants (or *constraints*) have originally been introduced by Liskov and Wing [23]; here, we will use the definition by Leino and Schulte [22] of history invariants as reflexive and transitive two-state invariants, relating any earlier to any later state of an object during a program's execution. Intransitive two-state invariants (as used e.g. by Cohen *et al.* [2] for their work on concurrent programs), relating only *successive* states, are unfit for our purposes, because they would be redundant with the pre-/postcondition and insufficient for reasoning about multiple (perhaps even an unknown number of) closure invocations.

Returning to our example, we can specify `old(count) <= count` as a history invariant for `inc`:

```
let mut count: i32 = x;
let mut inc =
    // view: count: i32 = count
    // invariant: old(count) <= count
    // ensures: result == old(count) && count == old(count) + 1
    move || { let r = count; count += 1; r };
```

Note that `old()`, when used in an invariant definition, does not refer to an *immediate* prestate, as it would in a postcondition; rather, as noted above, `old()` refers to *any* former state (and, indeed, thanks to the reflexivity requirement, this could even be the current state). This allows us to prove the example from above, by instantiating the history invariant for our concrete pair of states:

```
// label L1
let a = inc ();
// label L2
// ...
// label L3
let b = inc ();
assert! (a < b);
```

We can now reason that `a == old[L1](inc.count)` (with `inc.count` referring to the view `count` of `inc`) and `old[L2](inc.count) == old[L1](count) + 1`. Furthermore, by the history invariant, we know that `old[L2](inc.count) <= old[L3](inc.count)`; and so `b == old[L3](inc.count)` gives us `a < b` as desired.

Preservation of History Invariants Across Assignments

There is one remaining challenge for working with history invariants, which is exemplified by this piece of code:

```
let hocl = |i: i32| {
    let mut count = i;
    return move || {
        let r = count;
        count += 1;
        return r;
    };
};

let mut a = hocl (5); // invariant old(count) <= count (count == 5 here)
let mut b = hocl (1); // count == 1
a = b; // history invariant violated
```

Although we haven't defined a syntax for this situation yet, assume `hocl` guarantees that its result will have a view[3] `count: i32` and an invariant `old(count) <= count`. The assignment `a = b;` is valid only because the instances `a` and `b` both originate from the same definition (if not, they would have different (and thus unassignable) types, as every closure definition gets a unique, anonymous type in Rust). This means that we can only ever assign instances of the same closure definition to each other, which means that they will always have the same specification, but they may differ in their captured state, and so assignments can violate history constraints.

However, is the history invariant *really* violated? After all, one can argue that after the assignment, `a` contains a different instance, and that invariants should only be preserved along the lifetime of the same instance. The problem is that "objects" (such as closure instances) in Rust do not have a clear *identity*, meaning it is not actually possible to check whether a variable still holds the same instance or not—testing for equality (if it were allowed; closure instances can't be tested for equality in Rust, but the same issue applies to other aggregate types, like `struct`s) would simply compare the values of the captured state, but not any notion of identity.

Further difficulties arise when working with references: Assume that we borrow `a` mutably, and pass the reference to a higher-order function; how do we know whether `a` still holds the "same" instance afterwards, and so, in particular, whether the history invariant has been preserved? Without this knowledge, history invariants are rather useless because we never know whether or not they have been preserved whenever passing closures around; but passing closures around is precisely one of the use cases that motivate history invariants in the first place.

To resolve this conundrum, we propose a special operator `hist_inv(T, T) -> bool`, with `T` being some closure type, that returns true *iff* the history invariant holds between the two arguments. This is automatically true for every closure call (i.e., between the old and new instance value) but must be preserved explicitly otherwise. For instance, in the example above, just before the assignment, `hist_inv(a, b)` is false but `hist_inv(b, a)` would be true.

A higher-order function could require this in its precondition:

[3] Actually, this will be solved with *ghost result functions* instead of views; see Section 4.4.2, where we will revisit this very example in Figure 4.2.

```
// requires: hist_inv(*f, g)
// ensures: hist_inv(old(*f), *f)
fn foo<T: FnMut () -> i32> (f: &mut T, g: T)
{
    *f = g;
}
```

Here, `foo`'s specification says that `f`'s history invariant is preserved across the call to `foo`. In fact, considering how the *vast* majority of higher-order functions never assign any closure instances to each other, it makes sense to add a default "invariant preservation" postcondition of the form `hist_inv(old(*f), f)` to every higher-order function, for every mutable argument closure reference `f` that it takes. A higher-order function that *does*, in fact, "break" history invariants by assigning "incompatible" instances could be marked with a special `breaks_invs` annotation, which would compel the verifier to refrain from adding these defaults.

Invariants for Boxed Closures

Say `cl` is a boxed closure (of type `Box<dyn Fn...>`). Then an expression of the form `hist_inv(old(cl), cl)` would not be well-formed, because the box could have been reassigned to a completely different instance, meaning `old(cl)` and `cl` do not refer to instances of the same definition, and thus may have different captured variables, different invariants, *etc.* We must therefore adjust our conception of `hist_inv` when the argument has a dynamic type (such as `dyn Fn () -> i32`; either in a box or as a dynamic reference): Specifically, `hist_inv(a, b)` should hold *iff* both `a` and `b` have the same concrete type (at runtime), *and* `hist_inv(a, b)` holds for `a` and `b` as their concrete type.

4.1.4. Ghost State for "Tracing"

Consider the following example, where we define a closure that returns a unique identifier:

```
let count = 0;
let id =
    // view: count: i32 = count
    // invariant: old(count) <= count
    // ensures: result == count && count == old(count) + 1
    || -> i32 { count += 1; count };
```

This specification works in that we can prove that every returned value is, indeed, unique (because we know the invariant `old(count) <= count` and the postcondition `count == old(count) + 1`, and thus the result is always strictly larger by at least one than all previous values of `count`). However, the specification does not *straightforwardly* express this, and it also violates information hiding, because it exposes too much about the implementation. For instance, the following implementation should be equally correct for our purposes:[4]

```
let mut vals = vec! [];
let mut id2 =
    // view: vals: Vec<i32> = vals
```

[4] At least when reasoning about partial correctness, because this version is no longer guaranteed to terminate.

```
    // invariant: old(vals) subset vals
    // ensures: result in vals && !(result in old(vals))
    || -> i32 {
        let mut r: i32;
        while { r = rand::random (); !vals.contains (&r) } {}
        vals.push(r);
        r
    };
```

What we really want to express is that the returned value has never previously been returned, which is similar to what the previous example guarantees in its postcondition, but we want a *built-in* way to express this, so that *both* the former *and* the latter implementation will adhere to the same specification.

For this, we propose additional ghost state `prev_args`, which is a multiset of tuples, corresponding to the closure's signature, containing, conceptually, one tuple for every call to the closure, with the arguments of that call (*deeply*, i.e. containing not only references, but also their contents). This allows us to describe the captured state more precisely; for instance, we can now express what a counter actually does:

```
let mut count = 0;
let inc =
    // view: count: i32 = count
    // invariant: count == |prev_args|
    || -> i32 { let r = count; count += 1; r };
```

where `|prev_args|` is the cardinality of `prev_args`, which contains only empty tuples here, as the closure does not take any arguments.

Here is a slightly more complex example, using set comprehensions to describe how the view `s` changes over time:

```
let mut s = 0;
let add =
    // view: s: i32 = s
    // invariant: s == sum { i for (_, i) in prev_args }
    |a: i32, b: i32| -> i32 { s += b; s + a };
```

Similarly, `prev_results` contains all previous results, which allows us to give a precise and abstract specification for both of the "unique identifier" implementations above, and does not require any exposure of the captured state:

```
let mut id =
    // ensures: !(result in old(prev_results))
    || -> i32 { ... };
```

Note that we don't need to add `result in prev_results` to the postcondition, as this is implicit in the meaning of `prev_results`. To prove the above specification, we might need some additional invariants:

```
let count = 0;
let id =
```

```
// view: count: i32 = count
// invariant: old(count) <= count
// invariant: forall r: i32 :: {r in prev_results}
//     r in prev_results ==> r <= count
// ensures: !(result in old(prev_results))
|| -> i32 { count += 1; count };
```

Of course, *actually* keeping track of all previous arguments and results would be quite heavyweight. But this is not usually necessary; as with the other invariants described in the preceding sections, their value lies mostly in instantiations for specific states; for instance:

```
let a = id ();
let b = id ();
assert_ne! (a, b);
```

We don't actually need to know all previous results of `id` here, only that at the time of the second call, `a` was in `prev_results` but `b` wasn't; that's sufficient to assert inequality.

We can also use this feature to express that a closure was *only* called a certain number of times; for instance, `map()` could guarantee something like `f.prev_args == old(f.prev_args) union { (x,) for x in old(self.vals) }` for its argument closure `f`. This allows for even more precise reasoning about the closure's captured state at the higher-order function's call-site.

4.2. Specification Entailments

Specification entailments are primarily motivated by two applications. First, higher-order functions need a way to express a "lower bound" on the behavior of its argument closures; for instance, `map()` will call its argument closure once for every element of the collection it is operating on, and thus, the precondition of the argument closure should hold *at least* for all of these elements as arguments. Similarly, a higher-order function might require that its argument closure's postcondition implies *at least* that the result is, say, non-negative. Thus, we need a way for higher-order functions to express such constraints on the behavior of their argument closures.

Furthermore, any concrete closure that is passed into such a higher-order function will usually have a stronger, more precise specification. Thus, we also need a way to check whether the concrete specification satisfies the "lower bound" given by the higher-order function's specification—in other words, we need a way to check whether the latter is a valid *weakening* of the concrete specification. Both of these purposes will be served by specification entailments.

Second, we need a way to restrict, or *strengthen*, closure specifications in order to take additional call-/instantiation-site information into account. Consider this example:

```
let create_cl = |i: i32| {
    move || i
};
let mut f = create_cl (1);
let mut g = create_cl (2);
```

Assuming we add proper specification annotations to this snippet, we'd like to be able to prove `f |= || { ensures: result == 1 }` and `g |= || { ensures: result == 2 }`, even though `f` and `g` are both instances of the same definition. Therefore, specification entailments should also serve the purpose of concretizing specifications, of "tying" them to specific instances. Note, however, that this creates a framing problem:

```
f = g;
assert_eq! (f (), 1); // fails
```

In other words, our knowledge about `f`'s behavior was invalidated by the assignment (in fact, we know the new behavior, i.e. that `f` is going to always return 2 from now on). At the same time, we want entailed specifications to be framed across calls, otherwise the specification would not be of much use; but calling a closure can also change its captured state, i.e. the instance value. This conundrum will be resolved in Section 4.2.3 by introducing history invariants and using them for a precise definition of when a specification is framed and when it is not.

The main idea behind specification entailments has already been discussed in Section 2.3.2. In the remainder of this section, we build on and extend Hader's work in several respects.

4.2.1. Components of `|=`

This is the basic syntax for specification entailments:

```
cl |= |a1: T1, a2: T2, ...| { requires: P, ensures: Q, invariant: I }
```

where

- `cl` is the closure instance whose behavior we are describing;
- `|a1: T1, a2: T2, ...|` is a list of binders for the closure arguments, whose types may be given explicitly but could also be inferred from the type of `cl`;
- `P` is an assertion representing the precondition;
- `Q` is an assertion representing the postcondition; and
- `I` is an assertion representing the invariant.

The closure instance could be any expression that evaluates to a closure (or function) type. The pre- and postcondition may refer to the arguments (and to the variables of the enclosing scope through `outer()`, as described in Section 4.2.5), and the postcondition may additionally refer to `result` and `old()`. The invariant will be described in Section 4.2.3. The arguments `a1`, `a2`, ... must not shadow variables bound in the enclosing scope, to avoid confusion and ambiguities.

4.2.2. Entailments in Conjunctions, Pattern Matches, and under Quantifiers

Traditionally, a function will have exactly one specification. Similarly, in Section 4.1, every closure was annotated with one specification.[5] But any closure or function usually adheres to more than one specification: Most specifications can be *strengthened* and *weakened*. Strengthening means adjusting the specification to match the closure's behavior more precisely, which requires full knowledge about the closure implementation and is not always desirable, even if that knowledge is available, as it may expose too many implementation details.

Weakening, on the other hand, does *not* require any additional knowledge about the closure: Anybody who knows that some specification holds for the closure may weaken it by strengthening the pre- and weakening the postcondition, without needing any additional information. Additionally, specification weakening is "harmless" in that it won't interfere with information hiding. And finally, specification weakening is necessary when passing closures around, e.g. as arguments to higher-order functions: Any concrete closure we pass into a higher-order function will have its own specification, possibly stronger than what is required by the higher-order function's precondition; thus, the closure's specification must be weakened in order to prove the precondition.

We can also indirectly weaken a specification by putting the specification entailment operator under an implication or pattern match, as in this example, which shows a possible precondition for the `map()` function of `Option<T>`:

```
// requires: match self {
//     None => true,
//     Some(x) => cl |= |i| { requires: i == outer(x) }
// }
fn map<U, F: FnOnce(T) -> U> (self, cl: F) -> Option<U> { ... }
```

Putting the specification entailment under the pattern match is a relaxation of the precondition: The caller only has to prove the entailment if the option actually contains (or may contain) a value. Similarly, different specification entailments could be given for the different branches of the `match`. Also, note how this pattern match is conceptually very similar to two conjoined implications; informally speaking:

```
// requires: (self is None ==> true)
//     && (self is Some(x) ==> cl |= |i| { requires: i == outer(x) })
```

Entailments in consequents of implications are therefore also supported. This can be useful when learning something retroactively about the closure's captured state:

```
// ensures: (result == 0) || (result == 1)
fn coin_toss () -> i32 { rand::random () % 2 }

fn foo () {
    let i = coin_toss ();

    let mut x = 0;
    if i % 2 == 0 { x = 3; } else { x = 5; }
```

[5] Technically, every closure was annotated with *at most* one specification, because pre- and postcondition can default to `true` if left unspecified, but this still yields exactly one specification per closure definition.

```
    let cl =
        // view: x: i32 = x
        // invariant: x == old(x)
        // ensures: result == 2*x
        move || { 2 * x };

    if i == 0 {
        assert_eq! (cl (), 6);
    }
}
```

Instead of exposing x (as in this example) and carrying that information around, we can write, after the closure definition (informally speaking, because actual Rust assertions, of course, take Rust expressions and thus don't support specification entailments):

```
assert (i == 0 ==> cl |= || { ensures: result == 6 })
       && (i == 1 ==> cl |= || { ensures: result == 10 });
```

This example also demonstrates the utility of conjoining specifications. Conjoined specification entailments also occurred in Figure 2.1:

```
assume cl |= |i: i32| { requires: i >= 0, ensures: result > 0 }
assume cl |= |i: i32| { requires: i < 0, ensures: result > 0 }
assert cl |= |i: i32| { requires: true, ensures: result > 0 }
```

Our methodology explicitly aims to account for and support such scenarios.

Universal quantification over specification entailments, too, can prove highly useful, as we will see in Section 4.4.4 and Figure 4.3 in particular. For now, we have to content ourselves with this contrived example, which nonetheless illustrates the idea:

```
assert forall k: i32 :: cl |= |i| { requires: i >= outer(k),
                                    ensures: result >= 2*outer(k) }
```

Although of perhaps a lesser practical utility, our encoding will also, in principle, allow for specification entailments to occur in disjunctions, under existential quantifiers, and even in negations and antecedents of implications. Decisions on the extent to which these should be supported in practice thus become a language design choice, rather than being constrained by technical limitations.

4.2.3. Entailments with Invariants

Entailing Invariants

Recall the discussion of invariants from Section 4.1.3, and refer to Section 4.4 for our method of exposing captured state. It is useful to be able connect the two, i.e. we want a way to express invariants on exposed parts of the captured state in specification entailments.

To illustrate, consider this contrived example of a higher-order function taking a counter closure:

```
// ghost_arg: cnt: (T) -> i32
// requires: c |= || { ensures: result == old(cnt(self))
//                                 && cnt(self) == old(cnt(self)) + 1 }
fn take_counter<T: FnMut () -> i32> (c: T) {}
```

This specification implies that `cnt(c)` is non-decreasing. But we can make this explicit by including an invariant in the specification entailment in the precondition:

```
// requires: c |= || { ensures: result == old(cnt(self))
//                              && cnt(self) == old(cnt(self)) + 1,
//                     invariant: old(cnt(self)) <= cnt(self) }
```

From `take_counter`'s perspective, this invariant has the standard meaning (as in Section 4.1.3), i.e. we can instantiate it for every concrete pair of states and so on. To prove it, `take_counter`'s caller can use her knowledge about the definition of `cnt` and `c`'s actual invariant to see whether the invariant in the specification entailment actually holds.

We can also use `prev_args` and `prev_results` from Section 4.1.4, which always have the implicit invariants of being non-shrinking, i.e. every previous value is a submultiset of every later value. For example, a higher-order function taking a unique identifier closure could be specified as follows:

```
// requires: c |= || { ensures: !(result in old(prev_results)) }
fn take_id<T: FnMut () -> i32> (id: T) { ... }
```

Entailments using Invariants

Consider, once again, the counter example:

```
let mut count = 0;
let inc =
    // view: count: i32 = count
    // invariant: old(count) <= count
    // ensures: result == old(count) && count == old(count) + 1
    move || -> i32 { let r = count; count += 1; r };
```

Now suppose we call `inc` once. Afterwards (with `inc.count == 1`), should the following entailment be considered to hold?

```
inc |= || { ensures: result >= 1 }
```

It certainly does not hold in all states; indeed, the previous, first call to the closure returned `0` and therefore violated this specification. On the other hand, taking the history invariant into account, we know that *from now on*, all calls will certainly return a value not less than one.

We can therefore refine our understanding of closure specifications and specification entailments to talk only about *reachable* states: Knowing the current value of `count`, namely `1` in our example, and the history invariant `old(count) <= count`, we can prove the entailment above, for all *future* calls. Note, however, that assignments can interfere with this:

```
// count and inc defined as above
let mut save = inc;
inc ();
// inc |= { ensures: result >= 1 }
inc = save;
assert_eq! (inc (), 0);
```

So how do we know whether an assignment preserves an entailed specification? The answer lies in the `hist_inv` operator from Section 4.1.3: For any two instances `cl1` and `cl2` of the same closure type, knowing `cl1 |=` ⟨spec⟩ implies `cl2 |=` ⟨spec⟩ *if* we can prove `hist_inv(cl1, cl2)`.

Entailments for Single Calls: `|=!`

We can refine the entailment operator even further: Instead of reasoning about all calls "from now on", sometimes it makes sense to talk about the very next call only. For this purpose, we define a special operator `|=!`, which checks whether the next call to the closure fulfills a certain specification.

To illustrate, consider the `Option::map()` function from Rust's standard library:

```
pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Option<U> {
    match self {
        Some(x) => Some(f(x)),
        None => None,
    }
}
```

Note, in particular, how `map` calls its argument closure at most once; thus, its precondition should only have to reason about the single next call to the argument closure `f`. This can make a difference for examples such as this one:

```
let mut count = 0;
let mut cl =
    // view: count: i32 = count
    // requires: i != count
    // ensures: count == old(count) + 1
    |i: i32| -> f32 { let r = 1.0 / ((i - count) as f32);
                      count += 1; r };

let opt = Some (1);
let a = opt.map (&mut cl); // works
let b = opt.map (&mut cl); // fails, precondition of cl does not hold
```

Here, `cl` can only be passed *once* to `opt.map()`, because the second call would violate `cl`'s precondition `i != count`. We can express the precondition for `map` using the `|=!` operator:

```
// requires: match self {
//               Some(x) => f |=! |i| { requires: i == outer(x) }
//               None => true
//           }
```

If `map`'s precondition had instead been expressed using the `|=` operator, even the first call would have been invalid, because `i == outer(x)` is *not* sufficient to imply `cl`'s precondition in *all* states; it *is*, however, sufficient to conclude that the precondition will hold for the very next call (only), i.e. with the captured variables in the *current* state (with `cl.count == 0`).

4.2.4. Nested `|=`

It is possible to nest specification entailments. The most obvious application for this is for higher-order closures:

```
// requires: g |= || { ensures: result >= 0 }
// requires: f |= |x| { requires: x |= || { ensures: result >= 0 } }
```

```
fn foo<G: Fn () -> i32, F: Fn (G) -> i32> (f: F, g: G) -> i32 {
    f (g)
}
```

This precondition for `foo` expresses that `g` has to return a nonnegative value and that we must be able to call `f` with any closure (with the correct signature) that returns only nonnegative values.

However, nested entailments are more generally useful than that. Consider an "immediate" function composition ("immediate" meaning that instead of returning the composition, the result for a given input is returned):

```
fn compose_imm<A, B, C, F, G> (mut f: F, mut g: G, a: A) -> C
    where F: FnMut (A) -> B,
          G: FnMut (B) -> C
{
    g (f (a))
}
```

What we need to express in the precondition of `compose_imm` is that we can call `f` with argument `a` and `g` with whatever result `f` produced. We can express this using nested entailments:

```
// requires: f |=! |x| { requires: x == outer(a),
//     ensures: outer(g) |=! |b| { requires: b == outer(result) } }
```

Remember that preconditions are contravariant, so writing `requires: x == outer(a)` does *not* mean that `f` can *only* be called with this argument; instead, it means that knowing `x == outer(a)` must be *sufficient* to imply `f`'s precondition. For instance, assume `A` is `i32` and `compose_imm`'s caller passes `42` in `a`. Then `f(a)`'s actual precondition could be `true`, `a > 0`, `a % 2 == 0`, ...

The same strategy also works for `g`'s precondition: Instead of stating it explicitly (which is impossible to do modularly, because we don't know which concrete `g`s will be passed into `compose_imm`) or with the help of ghost specification arguments (which is inelegant and causes extra specification overhead), we can write that "knowing `g`'s argument was the result of the call to `f` must be sufficient to imply `g`'s precondition". This is modular as well as powerful and flexible: For instance, assume `f(a)`'s postcondition implies `a % 2 == 0 ==> result > 0`, and that `compose_imm`'s caller passes `42` in `a`, then a valid argument for `g` would be any closure whose precondition requires that its argument is positive, despite the fact that `f` does not *in general* guarantee positivity of its result.

4.2.5. The `outer()` Keyword

The purpose of the `outer()` keyword is to relocate the evaluation of its argument subexpression into the context of the specification entailment itself. For instance, in the following example, we want to express that we can call `f(i)`, with the value of `i` from `foo`'s prestate:

```
// requires: f |= |x| { requires: *x == *i } // WRONG
fn foo (f: impl Fn (&mut i32) -> i32,
        i: &mut i32) -> i32 {
    f (i)
}
```

What is the meaning of `*i` in this example? Remember that `|=` contains an implicit quantification over states: It checks that the specification holds in *all* states.[6] In particular, then, writing `*x == *i` above is misleading, because we *don't* want to quantify over the state of `*i`; rather, we want `*i` to be evaluated in the context of the `|=` itself—namely, `foo`'s prestate. This is made explicit by writing `outer(*i)`:

```
// requires: f |= |x| { requires: *x == outer(*i) }
fn foo (f: impl Fn (&mut i32) -> i32,
        i: &mut i32) -> i32 {
    f (i)
}
```

Plain `*i` is forbidden because it is misleading and, depending on how the ambiguity of its meaning is resolved, either redundant with `outer(*i)` (if it is interpreted as being evaluated in the outer state) or meaningless (if it is interpreted in the inner state, where its value is unknown). The expressions inside the braces of the specification entailment may thus only refer to variables bound by the argument list (`|x|` in the example above) and literals; everything else needs to be wrapped in `outer()` to pre-empt ambiguities.

Entailments can also occur in a higher-order function's postcondition, as in this example:

```
// ensures: result |= || { ensures: result == outer(*i) }
fn bar (i: &mut i32) -> impl Fn () -> i32 {
    *i += 1;
    let val = *i;
    return move || val;
}

// ensures: result |= || { ensures: result == outer(old(*i)) }
fn baz (i: &mut i32) -> impl Fn () -> i32 {
    let val = *i;
    *i += 1;
    return move || val;
}
```

Note the subtle difference between the two specifications: One evaluates `*i` in the context of `|=`, which occurs in the higher-order function's postcondition, thus referring to `bar`'s poststate value of `*i`, whereas the other refers to `old(*i)` in the outer (`baz`'s) poststate, which yields the outer prestate.

4.2.6. `|=` Across Multiple Calls

Consider the `sort_by` higher-order function, which receives a comparator closure as a parameter that is supposed to implement an ordering relation that is reflexive, antisymmetric, and transitive. Expressing reflexivity is straightforward:[7]

```
// requires: cl |= |a, b| { ensures: a == b ==> result }
```

[6] Notwithstanding the extension of `|=` with history invariants from Section 4.2.3.

[7] We will assume in this section, for simplicity, that the closure won't modify its arguments (which could be mutable references). Of course, it would be easy to add this as an explicit postcondition: `a == old(a)` *etc.*

Antisymmetry and transitivity, however, aren't as straightforward. The problem is that these are properties that talk about *pairs* and even *triples* of calls, rather than individual calls; thus, the meaning of specifications as an implicit quantification over all possible calls is insufficiently expressive here.

To solve this, we propose a natural extension of this implicit quantification to allow for quantification over more than one state; for instance:

```
// requires: cl |= |a, b|, |c, d| {
//     ensures: result0 && result1 && a == d && b == c
//         ==> a == b }
```

Here, we provide *two* sets of binders (`|a, b|` and `|c, d|`) for the closure arguments; one for the first call and one for the second call (which implies that we would like to quantify over two calls here). In the postcondition, we have to distinguish the two results from the two calls, which may, of course, differ—`result0` will refer to the result of the call with the arguments `(a, b)`, and likewise for `result1`. The meaning of this example is to express the property of antisymmetry ($\forall a, b : a \leq b \land b \leq a \implies a = b$ for some binary relation $\leq$); transitivity can now also be expressed using three sets of argument binders and thus quantifying over three calls at once.

Somewhat more esoterically, we can even quantify over calls to different closures:

```
// requires: f, g |= |x|, |y| { ensures: result0 >= result1 }
fn foo (f: impl FnMut (i32) -> i32, g: impl FnMut (i32) -> i32) { ... }
```

This expresses that for any possible pair of calls to `f` and `g`, `f` will always return a result not less than `g`'s result. Note that we wouldn't be able to call `f` and `g` directly in the specification, because they may be impure.

4.2.7. Boxed Closures, Dynamic References to Closures, and Function Pointers

Let us now revisit the different ways of storing closures from Section 3.4. We will focus on boxed closures here, but the same principles should be applicable to dynamic references and function pointers as well.

Recall that our definition of specification entailment always involves some specific instance, and an entailment may not be preserved across an instance assignment if the history invariant is violated. This careful definition will prove very useful now, because specification entailments for boxed closures will also be preserved as long as no assignment takes place. Assignments can invalidate specification entailments whenever the history invariant is violated, as in Section 4.2.3:

```
let hocl = |i: i32| { move || i };
let mut f: Box<_> = Box::new (hocl (1));
let mut g = hocl(2);
// assert *f |= || { ensures: result == 1 }
*f = g;
assert_ne! ((*f) (), 1);
```

In fact, this example could be verified with the techniques presented so far: Let `T` denote the return type of `hocl`, then `f` will have type `Box<T>`, meaning we can only assign different instances of the

same definition to `f`, as before. But in general, and what makes boxed closures more interesting and challenging from a verification perspective, boxes can be reassigned to instances of completely unrelated definitions. Note the type of `f` in the following example, which denotes a box containing *some* statically-unknown (in general) subtype of `FnMut () -> i32`:

```
let mut f: Box<dyn FnMut () -> i32> = Box::new (|| 42);

let mut count = 0;
let cl = move || { let r = count; count += 1; r };
f = Box::new (cl);
```

Recall our definition of `hist_inv()` for dynamic types from Section 4.1.3: `hist_inv(a, b)` can only hold if both `a` and `b` are instances of the same definition. In particular, then, the assignment to `f` above invalidates all specification entailments known to hold for the old (overwritten) value of `*f`. This aligns nicely with our previous definition of entailments—namely, that entailments are preserved as long as `hist_inv(`*old instance*`,` *new instance*`)` holds.

Similarly, knowledge about the new value should be preserved; for instance:

```
let mut f: Box<dyn Fn () -> i32> = Box::new (|| 1);
let g: Box<dyn Fn () -> i32> = Box::new (|| 2);
// assert *f |= || { ensures: result == 1 }
// assert *g |= || { ensures: result == 2 }
f = g;
// assert *f |= || { ensures: result == 2 }
```

This approach also works in the presence of nondeterminism:

```
let mut f: Box<dyn FnMut () -> i32>;

if coin_toss () {
    f = Box::new (|| 42);
    // assert *f |= || { ensures: result >= 0 }
} else {
    let mut count = 0;
    f = Box::new (
        // view: count: i32 = count
        // invariant: count >= 0
        // ensures: result >= 0
        move || { count += 1; count });
    // assert *f |= || { ensures: result >= 0 }
}

// assert *f |= || { ensures: result >= 0 }
```

Given that both branches of the `if`/`else`-block establish the desired entailment for `*f`, it should hold at the join point after the `if`.

Boxed Closures as Struct Fields

So far, we have been arguing that it is necessary to keep track of whether `hist_inv(old_cl, cl)` holds between some old instance value `old_cl` and the current value `cl` in order to preserve entailments known to hold for the old value.

To preserve modularity, higher-order functions have to guarantee this in their postcondition; this could be made the default in the implementation:

```
// ensures: hist_inv(old(**f), **f)
fn foo (f: &mut Box<dyn Fn () -> i32>)
{
    f ();
}

let mut f: Box<dyn Fn () -> i32> = Box::new (|| 1);
// assert *f |= || { ensures: result == 1 }
foo (&mut f);
// assert *f |= || { ensures: result == 1 }
```

This is impractical for `struct`s, however. Consider this example (assuming history invariants are also supported for `struct`s):

```
struct StoreCl {
    // invariant: hist_inv(old(cl), cl)
    cl: Box<dyn Fn () -> i32>
}
```

By itself, this does not tell us anything about `cl`'s behavior, because we do not know the concrete closure type stored in `cl` and thus cannot know what `hist_inv` even implies. Additionally, without knowing this concrete type, we also cannot reassign the `cl` field, because there would be no way to prove `hist_inv(old(cl), cl)`. Thus, this invariant on the `StoreCl` type is not particularly useful; it would make more sense to track `hist_inv` as before (e.g. `hist_inv(old(s.cl), s.cl)` for some instance `s` of `StoreCl`).

Instead, we can give a specification entailment as a (single-state) invariant:

```
struct StoreCl {
    // invariant: *cl |= || { ensures: result >= 0 }

    cl: Box<dyn Fn () -> i32>
}
```

This way, the specification entailment becomes part of the `StoreCl` type, and preservation is guaranteed because every assignment to `cl` will have to maintain the type invariant for `StoreCl` and, therefore, the entailment. At the same time, this invariant is useful even without knowing the concrete type of `*cl`, and it is flexible enough to allow for reassigning `cl`, as long as the specification entailment is preserved.

For a more complex example and a pragmatic discussion of different specification approaches for this situation, refer to Section 6.1.4.

4.3. Arrow (~~>) Notation

To describe the effects of a higher-order function modularly, without relying on or exposing the concrete specification of an argument closure, we present a novel notation which we shall refer to as the *arrow* notation, named for its central syntactic element, a squiggly arrow (`~~>`).

Consider the following example code:

```
// requires: f |= |x| { requires: *x == outer(*i) + 1 }
fn foo (f: impl Fn (&mut i32) -> i32,
        i: &mut i32) -> i32 {
    *i += 1;
    f (i)
}
```

What should the postcondition of `foo` be? We could add a postcondition to the specification entailment; for instance:

```
// requires: f |= |x| { requires: *x == outer(*i) + 1,
//                      ensures: result > 0 && *x > old(*x) }
// ensures: result > 0 && *i > old(*i) + 1
```

This would be non-modular and inflexible, though, because a client might want to pass in some `f` with a different behavior/postcondition (but couldn't with this specification). We can solve this by making the postcondition parametric, using a ghost argument, similar to Hader's work:

```
// ghost_arg: P: Pred(i32, i32, i32)
// requires: f |= |x| { requires: *x == outer(*i) + 1,
//                      ensures: P(old(*x), *x, result) }
// ensures: P(old(*i) + 1, *i, result)
```

As is evidenced by this small example already, this approach quickly becomes unwieldy, as the ghost predicate needs to relate more and more arguments in different states. It is also impractical to have to specify P manually (unless it could be inferred automatically, as already suggested in Section 2.3.2). Furthermore, this approach won't work for mutable captured state: Unless the captured state is exposed statically, which would again be inflexible and non-modular, `foo`'s specification doesn't know about and can't access the captured state. In particular, this means that `foo`'s specification cannot pass (parts of) the captured state to P; but then `foo`'s specification would be too weak to reason about how the captured state has changed during `foo`'s execution. It would be possible, though, to extend the specification language in such a way as to allow `f`, the closure instance, to be passed to the ghost predicate directly, and to allow `foo`'s caller to access the captured state when defining P.

Instead, let us consider the following properties we want to express about `foo`'s behavior, which illustrate the needs that our specification language must address:

1. `f` was called with argument `i`,
2. in a state where `*i == old(*i) + 1`; that
3. `foo`'s result is whatever result `f` produced, and

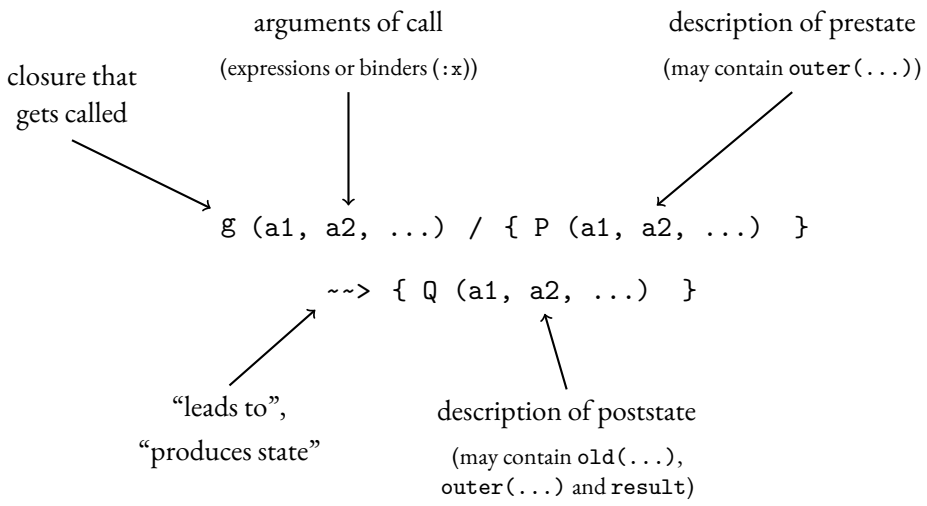

Figure 4.1.: Components of the arrow notation.

4. `i` is in the state that `f` left it in (in other words, `foo` didn't modify `*i` further after calling `f`).

To fulfill those requirements and capture all of this information, we have devised a novel notation, whose components are described in Figure 4.1; Section 4.3.1 will go into more details for each of the components. But as a preview, let us look at a formalization of the points described above:

```
f (:k)                              // f was called with some k: &mut i32
/ { *k == outer(old(*i)) + 1 }      // in a state where *k == old(*i) + 1
~~>                                 // producing a state where
{ outer(result) == result           // foo's result equals f's result, and
  && outer(i) == k }                // i is equal to k in f's poststate
```

This belongs into `foo`'s postcondition:

```
// requires: f |= |x| { requires: *x == outer(*i) + 1 }
// ensures: f (:k) / { *k == outer(old(*i)) + 1 }
//     ~~> { outer(result) == result && outer(i) == k }
fn foo (f: impl Fn (&mut i32) -> i32,
        i: &mut i32) -> i32 {
    *i += 1;
    f (i)
}
```

4.3.1. Components of ~~>

Closure that Gets Called

Any expression that evaluates to a closure (or function) type is valid here. It will be evaluated in the *current* state, meaning in the context enclosing the `~~>`. For instance:

```
// requires: *f |= |i| { requires: i == 42 }
// ensures: old(*f) (42) / {} ~~> { outer(result) == result }
fn foo<T: Fn (i32) -> i32> (f: &mut T, g: T) -> i32 {
    let r = (*f) (42);
    *f = g;
    r
}
```

Replacing `old(*f)` with `*f` in `foo`'s postcondition would be wrong, because `*f` in `foo`'s poststate is different from `*f` when `f` was called. This is of particular relevance to `FnMut` closures, whose instance value (which includes the captured state) can change during every call:

```
// requires: *f |= || { requires: true }
// ensures: old(*f) () / {} ~~> { outer(result) == result }
fn bar (f: &mut FnMut () -> i32) -> i32 {
    f ()
}
```

Again, plain `*f` in `bar`'s postcondition (instead of `old(*f)`) would be wrong, because `*f` (i.e., the instance value, including the captured state) in the poststate is different[8] from `*f` at the point where `*f` was called.

When calling `FnMut` closures multiple times, we have to introduce an existential quantification to talk about intermediate closure instance values (for the use of `self` in this example, refer to Section 4.4.3):

```
// requires: *f |= || { requires: true }
// ensures: exists fi: T ::
//     old(*f) () / {} ~~> { self == fi }
//     && fi () / {} ~~> { self == outer(*f) }
fn baz<T: FnMut () -> ()> (f: &mut T) { f (); f (); }
```

This specification allows us to prove call-site assertions such as this one:

```
let mut count = 0;
let mut inc =
    // view: count: i32 = count
    // ensures: count == old(count) + 1
    || { count += 1; };
baz (&mut inc);
assert_eq! (count, 2);
```

Because quantifying over closure instances as in the previous example would be so commonly necessary in practice, we propose a shorthand notation `:f`, which refers to some (existentially-quantified) instance `f_` such that `hist_inv(old(f), f_)`. For instance, a `map()` on a collection can then have a postcondition similar to this:

```
// ensures: forall x: T :: old(self.vals).contains(x) ==>
//     :f (x) / {} ~~> { outer(result).contains(result) }
```

Note that this is unnecessary for `Fn` closures because their instance value does not change during calls (although they, too, can be reassigned, but this is not too common in practice).

[8] At least potentially; we do not know `f`'s concrete behavior at this point.

Arguments of the Call

Arguments are expressions (of appropriate types, as given by the type of the closure/function that gets called) that will be evaluated in the *current* state, but must evaluate to the values they had at the time of the call. The rationale for evaluating them in the current state is that the actual state in which the call happened is internal to the higher-order function and not known to the caller (except that it's "somewhere between" pre- and poststate). Thus, attempting to evaluate expressions in that state would be meaningless at the call-site for mutable locations. For instance, the following postcondition is valid and useful for `foo`'s caller, expressing that "`f` was called with some reference (deeply) equal to `i` in `foo`'s poststate":

```
// ensures: f (i) / {} ~~> { result == outer(result) }
fn foo (f: impl Fn (&i32) -> i32, i: &mut i32) -> i32 {
    *i += 1;
    f (i)
}
```

This creates a different issue, though: What if we can't name the reference that was passed to the closure? For instance:

```
fn bar (mut f: impl FnMut (&mut i32) -> i32) -> i32 {
    let mut x = 42;
    let y = f (&mut x);
    y + x
}
```

One might be tempted to try to express this function's behavior as "there exists `r: &mut i32` such that `*r == 42` and `f` was called with `r`". Prusti does not support quantification over references, though, and especially not over mutable references, which are "non-copy" types in Rust (i.e., they don't implement the `Copy` trait and thus must be moved instead of being copied). It is also unclear how quantification over references would work, especially when trying to evaluate the reference in multiple states.

To avoid all of these difficulties, we propose a custom variable binder syntax that binds a function argument for the express purpose of describing it in both the pre- and poststate of the call. The bound variables will be prefixed with a colon to mark them as bound; for instance, the `foo` function presented above could be annotated with a postcondition such as:

```
// ensures: old(f) (:r) / { *r == 42 } ~~> { outer(result) == *r + result }
```

This is *conceptually* equivalent to an existential quantification over `r`, but it is restricted to this specific use case of binding a function argument, and thus avoids the aforementioned difficulties incurred by quantifications over references in general.

For types that *can* be quantified over in Prusti, the binder notation is a convenient shorthand; for instance, the following two postconditions are equivalent:

```
// ensures: exists k: i32 :: k % 2 == 0
//     && f (k) / {} ~~> { outer(result) == result }
// ensures: f (:k) / { k % 2 == 0 } ~~> { outer(result) == result }
fn bar (f: impl Fn (i32) -> i32) -> i32
```

```
{
    let i = rand::random::<i32> () * 2;
    f (i)
}
```

Using this binder notation is necessary if a mutable reference is passed into a closure:

```
// ensures: f (:r) / { *r == outer(old(*i)) + 1 } ~~> { *r == outer(*i) }
fn foo (f: impl Fn (&mut i32) -> (), i: &mut i32) {
    *i += 1;
    f (i);
}
```

We could also do without the binder notation in some cases; for instance:

```
// ensures: f (old(i)) / {} ~~> {}
fn f1 (f: impl Fn (&mut i32) -> (), i: &mut i32) {
    f (i);
}
```

But then we would not be able to describe the final value of `i`, because we don't have a way of referring to `f`'s argument in the poststate description of `~~>`.

Prestate Description

The prestate description is an assertion that must evaluate to true in the function call's prestate. It may only refer to variables bound in the argument list of the arrow notation; variables from the enclosing scope must be wrapped in `outer()`, which is discussed in more detail in Section 4.3.2. The prestate description defaults to true if none is given.

Poststate Description

Similarly, the poststate description is an assertion that must hold in the call's poststate. The same restrictions apply as for the prestate description, although the poststate description may additionally use `old()` to refer to the call's prestate (not to be confused with `outer(old())`, which would refer to the enclosing higher-order function's prestate) as well as `result`, the value returned by the call (again, not to be confused with `outer(result)`).

4.3.2. The `outer()` keyword

The `outer()` keyword here has the same function and meaning as the one discussed in Section 4.2.5: It relocates the evaluation of its argument subexpression into the context enclosing the `~~>`. The expressions inside of the braces of the `~~>` syntax have the meaning of being evaluated in specific states, and so referring to variables from the enclosing scope can lead to confusion and is therefore not allowed except if wrapped in `outer()`.

Recall this higher-order function example from above:

```
// requires: f |= |x| { requires: *x == outer(*i) + 1 }
// ensures: f (:k) / { *k == outer(old(*i)) + 1 }
//      ~~> { outer(result) == result && outer(i) == k }
fn foo (f: impl Fn (&mut i32) -> i32,
        i: &mut i32) -> i32 {
    *i += 1;
    f (i)
}
```

Here, the `~~>` occurs in `foo`'s postcondition; thus, `outer(i)` will refer to `i` in `foo`'s poststate. Similarly, `outer(old(*i))` will also evaluate `old(*i)` in the context of `~~>`—i.e., as if it occurred in `foo`'s postcondition—and thus yield `*i`, evaluated in `foo`'s prestate. `outer(result)` is a special overload referring, in this case, to `foo`'s result: Referring to `f`'s result in `foo`'s poststate would not make sense because `foo` might call `f` many times, and so it would not be clear which result is meant.

To further illustrate the meaning of `outer()`, consider this example, where we use `~~>` to describe a call to `foo`:

```
fn bar (g: impl Fn (&mut i32) -> i32,
        k: &mut i32) -> i32 {
    foo (g, k)
}
```

`bar`'s postcondition could be expressed as follows:

```
// ensures: foo (g, :k1) / { *k1 == outer(old(*k)) } ~~> {
//              g (:k2) / { *k2 == outer(old(*k1)) + 1 }
//                  ~~> { outer(result) == result
//                        && outer(k1) == k2 }
//              && outer(result) == result
//              && outer(k) == k1
//          }
```

Here, we have a nested `~~>`, and thus, the innermost occurrences of `outer()` refer to the poststate of the call to `foo`, whereas the first and last two `outer()` refer to `bar`'s poststate.

But `outer()` may also refer to the prestate; for instance:

```
// requires: h (42) / { } ~~> { result == outer(*l) }
fn baz (h: impl Fn (i32) -> i32, l: &mut i32) -> ...
```

In this, admittedly obscure, example, `baz` requires that when it is called (i.e., in its prestate), `*l` must have a value that was produced by calling `h` with argument `42`. `outer()` thus refers to `baz`'s prestate, which is consistent with our definition, because `~~>` itself occurs in the precondition.

This construction can be particularly useful whenever one function needs to ensure the precondition of the next. We have already seen an example of this in Section 4.2.4, where we used a nested entailment to express that the first function must ensure that the second function can be called. Rather elegantly, we can also express the same fact from the second function's point of view: Instead of the first function ensuring the precondition of the next, the second function can *require* the postcondition of the first:

```
// requires: f |= |k| { requires: k == outer(x) }
// requires: g |= |m| {
//     requires: f (outer(x)) / {} ~~> { result == outer(m) } }
fn compose_imm<A, B, C> (
    f: impl Fn (A) -> B, g: impl Fn (B) -> C, x: A) -> C
{
    g (f (x))
}
```

Writing a suitable postcondition for `compose_imm`, too, is straightforward with the `~~>` notation:

```
// ensures: g (:b) / {
//              f (outer(old(x))) / {} ~~> { result == outer(b) }
//          } ~~> { result == outer(result) }
```

The arrow notation provides us with the means to reason about the effects of closure calls abstractly, without requiring any knowledge about their concrete behavior. Together with specification entailments, they serve as the foundation of our approach for reasoning about higher-order functions. We will now turn to solving the remaining problems of reasoning about captured state across function boundaries, and of describing complex functional behavior, such as that of `fold()`.

4.4. Ghost Arguments and Results

4.4.1. Basics Ideas

> Ghost code is a part of the program that is added for the purpose of specification. Ghost code must not interfere with regular code, in the sense that it can be erased without observable difference in the program outcome. [7]

We have already encountered ghost state in Section 4.1.4. Here, we want to extend our specification language with ghost *arguments* and *results*, i.e. values that are passed in and out of functions for verification purposes, but that don't actually occur anywhere in the code the compiler will generate.

As a simple example (inspired by the one from Filliâtre *et al.* [7]), consider this piece of code:

```
// pure
fn fib (n: u32) -> u32
{
    match n {
        0 => 0,
        1 => 1,
        k => fib (k - 1) + fib (k - 2)
    }
}

// ghost_arg: k: i32
// requires: a == fib (k) && b == fib (k + 1)
// ensures: result == fib (k + 2)
```

```
fn next_fib (a: u32, b: u32) -> u32
{
    a + b
}
```

Here, `next_fib` doesn't *actually* need to know that its arguments are the k-th and (k+1)-th Fibonacci number; this fact happens to be irrelevant for `next_fib`'s computation. But it *is* relevant for verifying the postcondition, so we can use a ghost argument. Ghost results work in the analogous way—instead of being passed *into* the function (conceptually, because the transfer doesn't actually take place, only for verification), they are passed *out of* the function.

4.4.2. Exposing Captured State via Ghost Arguments/Results

One useful application for ghost argument/result functions is for exposing captured state. Hader [12] relies on *views* (as described in Section 4.1.1) and even includes them in specification entailments, as follows:

```
// requires: f |= || { views: [mut cnt: i32],
//                     ensures: cnt == old(cnt) + 1 }
// ensures: f.cnt == old(f.cnt) + 1
fn foo (f: &mut impl FnMut () -> i32) { f (); }
```

This is confusing, because for `f.cnt` to even be well-formed, `f` has to fulfill the specification entailment (otherwise the view `cnt` may not even exist). This implicit dependency is not obvious and can further be complicated if the specification entailment occurs, say, under a quantifier or in an implication.

In addition, since Hader matches the views by name (i.e., the specification entailment requires `f` to have a view named `cnt` of type `i32` that the closure may mutate), this creates a structural subtyping situation: *Any* closure with a view named "`cnt`" could pass the specification entailment, even if that view might have nothing to do with the *intended* `cnt` (although this situation is hard to imagine for the simplistic example given here). And finally, this is non-modular, because the higher-order function should not depend on its argument closure having a particular view, of a particular name and type.

To solve all of these issues, we propose the use of ghost argument functions, which can take the closure instance as a parameter (for the use of `self` here, refer to Section 4.4.3):

```
// ghost_arg: cnt (T) -> i32
// requires: f |= || { ensures: cnt(self) == old(cnt(self)) + 1 }
// ensures: cnt(f) == old(cnt(f)) + 1
fn foo<T: FnMut () -> i32> (f: T) { f (); }
```

On the surface, it might look as though the two specifications are almost identical. However, the latter specification features several advantages by introducing this additional layer of abstraction between the closure's captured state and the higher-order function:

- The ghost argument is defined separately, so the specification entailment could be in a disjunction, under a quantifier, there could be multiple entailments, *etc.*, all without affecting the availability of `cnt`.

- The closure no longer has to have a view named `cnt`; it could have a view of the same type with a different name, or even something completely different, as long as the user can come up with a value for `cnt`. This also avoids the "accidental" structural subtyping situation.
- This solution is also more modular, because it forces the *user*, at the call-site, to think about the proper value for `cnt`. Otherwise, in Hader's solution, the closure *definition* needs to anticipate the higher-order function call and define the appropriate view(s), *and* the higher-order function definition needs to specify some name and type for the view, which it can't possibly know modularly.

This solution is also simpler, because we already need ghost arguments for Section 4.4.4 (and, in fact, Hader uses them, too), so we don't need any additional machinery.

Ghost *result* functions can also make sense, especially whenever a higher-order function *returns* a closure; to illustrate, let us revisit an example from Section 4.1.3, now with ghost result annotations, in Figure 4.2. Note that we must introduce a special keyword `ReturnT`[9] to refer to the *type* of the result (in the signature of `cnt`); the outer closure returns another closure, which has an anonymous type, so we can't write it down. In fact, even for functions, which, unlike closures, can have generic arguments in Rust, it is not possible to write down the returned closure type explicitly, because it is an existential type, whereas generic arguments are interpreted as universal types; for instance, the following function returns "some" type implementing `Fn () -> i32`:

```
fn return_cl () -> impl Fn () -> i32 { ... }
```

Note that for a meaningful definition of such ghost argument/result functions, it is necessary for their definitions to be able to access the captured state (i.e., the views), such as `t.c` in the definition of `cnt` in Figure 4.2. This is not a technical problem, because encoding closure types as (or similar to) `struct`s allows us to do this very easily, but it is nonetheless worth keeping in mind that this transcends Rust syntax, where accessing the captured state by any means from outside the closure body is not allowed.[10]

4.4.3. `self`: Accessing the Captured State Opaquely

Although not directly related to ghost arguments/results, this feature is necessary for and motivated (though not exclusively) by them. Recall this example from above:

```
// ghost_arg: cnt (T) -> i32
// requires: f |= || { ensures: cnt(self) == old(cnt(self)) + 1 }
// ensures: cnt(f) == old(cnt(f)) + 1
fn foo<T: FnMut () -> i32> (f: T) { f (); }
```

Here, in the specification entailment, `f` needs to pass "itself" to `cnt` (i.e., the closure instance denoted by `f` containing, in particular, its captured state). Writing `cnt(f)` would be awkward, because the specification doesn't depend on the *name* of the closure instance; in fact, if we assigned the instance, say by writing `let mut g = f;`, then the same specification should now hold for `g`. Furthermore,

[9] Given that the returned value can be referred to via the `result` keyword in specifications, a more obvious choice for this keyword might have been "`Result`"; however, this name is already taken by the `std::result::Result` type.

[10] Save for `unsafe` code.

```
let hocl =
  // ghost_result: cnt: (ReturnT) -> i32
  // ensures: result |= || { ensures: result == old(cnt(self))
  //                              && cnt(self) == old(cnt(self)) + 1,
  //                         invariant: old(cnt(self)) <= cnt(self) }
  |i: i32| {
      let mut count = i;
      let cl =
          // view: c: i32 = count
          // ensures: result == old(c) && c == old(c) + 1
          move || {
              let r = count;
              count += 1;
              return r;
          };
      // ghost_return: cnt(t) := t.c
      return cl;
  };
```

Figure 4.2.: An illustration of the utility of ghost result functions for exposing captured state.

writing `f` inside of the entailment is misleading because the closure instance referred to doesn't have to be the *current* instance, denoted by `f`; it could also be some *future* instance, because the specification is preserved across calls.

Thus, we propose using the `self` keyword for this purpose. Note that this *does* interfere with member functions of structs, i.e. if `foo` above was actually implementing a method for some struct in an `impl` block, `self` should refer to the other struct members; but this issue can be solved easily using the `outer` keyword discussed before: `self` in the specification entailment above refers to `f`, whereas `outer(self)` in the same position would refer to the object that `foo` is attached to. Note that `outer(self)` does *not* refer to the outer value of the closure instance; for this, one would have to write `outer(f)`, or whatever the name of the instance may be in the enclosing scope.

`self` can also be used as part of the arrow notation, not just in specification entailments; for instance,

```
// ensures: (old (f)) () / { cnt(self) == outer(old(cnt(self))) }
//     ~~> { cnt(self) == old(cnt(self)) + 1 }
```

would be another valid postcondition for `foo`. In fact, using `self` can make sense even without ghost functions:

```
// ensures: (old (f)) () / {} ~~> { outer(f) == self }
```

This tells the caller that the instance value of `f` in the poststate was achieved by calling `f` once.

```
// ghost_arg: inv: (MultiSet[i32], i32) -> bool
// requires: inv({}, init)
// requires: forall prev_els: MultiSet[i32] ::
//     prev_els subset self.vals ==>
//       cl |= |c, a| { requires: inv(prev_els, a) && c in outer(self).vals,
//                      ensures: inv(prev_els union {c}, result) }
// ensures: inv(self.vals, result)
fn fold (init: i32, cl: impl Fn (i32, i32) -> i32) -> i32 { ... }
```

Figure 4.3.: A functional specification for `fold()`.

4.4.4. Ghost Arguments as Invariants

To verify the functional behavior of a `fold()`, say, or a custom `while`-loop implementation,[11] it is useful to work with *invariants*—just as with regular loops (accordingly, we use the term "invariant" in this section in the sense of "loop invariant", *not* as an invariant on the captured state, as in previous sections).

To verify its functional behavior, `fold()` could maintain some additional ghost state; namely, the multiset of elements seen so far. An example specification for `fold` is given in Figure 4.3. We will stick with `i32` in this section for simplicity, but of course the approach readily generalizes to a generic implementation.

Let us walk through this specification step-by-step:

- `inv` represents the invariant for the entire folding operation: It relates a multiset—the multiset of elements seen so far—and an integer, the current accumulated value, and returns `true` *iff* the accumulated value satisfies the invariant, given the multiset of elements seen so far. Note how this definition is more general than having `inv` take only the multiset and *return* an integer: The latter definition is "deterministic" (i.e., for every multiset, there can be only one accumulated value), whereas our definition allows for multiple different integers satisfying the invariant; for instance, as a contrived example, imagine we only care about the absolute value of the accumulated value, then both `inv(s, i)` and `inv(s, -i)` could return `true` for some `s` and `i`.

  As an example, say we use `fold` for summing up a sequence of integers—a very common operation in practice, along with other accumulations such as maxima/minima, products, *etc.*—then `inv(s, a)` could be defined as `sum(s) == a`, assuming the availability of a `sum` operator with the expected semantics.[12]

[11] Kassios and Müller actually discuss a custom `while`-loop implementation using closures and invariants in their paper. The difficulty for us is that they use two closures, one for checking the condition and one for the body—but in Rust, the body closure would most likely have to capture some variables mutably, and then the condition closure could not refer to those variables, because mutable borrows are exclusive. We have thus refrained from including the example here. (Actually, they use three closures, with a third closure for the invariant; but since the invariant is only needed for specification and proof purposes, it could be implemented as a ghost argument function in our model and thus wouldn't pose a problem. The condition can't be a ghost function, because it affects the program behavior.)

[12] While this is not currently available in Viper, previous research e.g. by Leino and Monahan [20] has demonstrated the feasibility of set comprehension operations in an automatic verifier.

- Requiring `inv({}, init)` to hold on entry is necessary for ensuring the initial value does, in fact, establish the invariant when no elements have been visited yet, i.e. for the empty set of elements seen so far.
- We do not want to depend on the order of operations in `fold`, as this would complicate matters heavily. Therefore, we put the specification entailment under a quantifier: For *any* subset of `self.vals` (assuming, for simplicity,[13] that this is the collection we are folding over), the closure may assume that the invariant holds with regard to this multiset and the current accumulated value, its argument `a`. Further, it may assume that its first argument, `c`, the current value, is in the collection we are folding over.[14] Then, it must ensure that the invariant holds for the enlarged multiset and its result, the new accumulated value.
- Finally, `fold` can guarantee that the invariant holds in the end for the entire collection and the resulting accumulated value.

Note that the invariant could also involve the closure instance; this would allow the argument closure, for instance, to accumulate a result in its captured state, in addition to or instead of the more explicit accumulation done by `fold()`.

4.5. Summary

This chapter has introduced a variety of techniques that work together to provide the means for verifying use cases from all of Chapter 3's four categories. Our aim has been to refine the different tools (specification entailments, history invariants, *etc.*) in such a way as to promote their integration and confluence, so that, first, the different techniques are orthogonal and can be combined and used together, and second, the techniques are powerful and flexible enough to work for all of the four classes of closure uses, instead of necessitating completely different techniques for different classes.

Section 6.1 will discuss the application of the techniques from this chapter to more complex, realistic examples. Before that, Chapter 5 will explain how to encode the methodology presented in this chapter in Viper.

[13] Usually, `fold` will work on iterators in Rust, but this is not important for our purposes here.
[14] We cannot guarantee `!(c in prev_els)`, because the collection could contain duplicates.

5. Implementation

In this chapter, we shall explore how the methodology described in Chapter 4 is implemented in our verification toolstack, described in Chapter 2. We will begin by examining how to encode the user-level closure and higher-order function specification constructs in Viper, an intermediate verification language, in Section 5.1; Section 5.2 will then discuss how to automate this translation as part of the Prusti front-end verification tool.

5.1. Encoding in Viper

We will now return to the specification functions from Section 2.3.1, adapted to our purposes and the Viper environment. Remember that we had to quantify over heaps and pass heaps to the specification functions in Section 2.3.1, which we deemed incompatible with our approach based on separation logic and Viper. Luckily, though, we do not need to reason about *entire* heaps when talking about closure calls in Rust, only the parts of the heap reachable (by following references, boxes, *etc.*) from the closure instance (which includes the captured state), the arguments, and the result; furthermore, for the instance value and the arguments, we need to distinguish their prestate and poststate values.[1]

We can capture all of this data by using *snapshots*,[2] which are mathematical abstractions of Rust types. They abstract heap-dependent values *deeply* into a `domain` type in Viper, which means that they are state-independent. They can be copied, passed around, *etc.*, all without worrying about permissions, states, and so on. We can even quantify over them easily, and/or use them in conjunction with pure functions in powerful ways, because reasoning about snapshots (e.g. for framing pure function results) is easier than reasoning about heap-dependent values, which depend on the current state, require certain permissions, *etc.* To illustrate, consider the example in Figure 5.1 of how a `struct X { a: i32 }` is modeled with a snapshot type `snap_X`, together with a conversion function `X_to_snap`, which "lifts" a reference to the snapshot type.

We now define the following specification functions per closure type `T`. All of them receive their arguments as snapshots, so that we can, among other things, quantify over them for the encoding of various constructs in the remainder of this section:

- `pre$T()`, taking the closure instance and all of its arguments as parameters, and returning `true` *iff* the precondition holds for the given closure instance and arguments.
- `post$T()`, taking the closure instance (in its poststate, conceptually), the old closure instance (from its prestate), all of its arguments and old arguments, and the result as parameters, and

[1] Global state is considered unidiomatic in Rust, as it interferes with the ownership system. In particular, all global variables should be immutable compile-time constants; mutable global variables are considered `unsafe`. [16, Section 19.1]

[2] Snapshots were added to Prusti by Christoph Matheja to facilitate reasoning about pure functions.

```
field int_val: Int
field f_X_a: Ref

predicate X (x: Ref) {
    acc(x.f_X_a) && acc(i32(x.f_X_a))
}

predicate i32 (i: Ref) {
    acc(i.int_val)
}

domain snap_X {
    function get_a (x: snap_X): Int
}

function X_to_snap (x: Ref): snap_X
    requires acc(X(x))
    ensures get_a(result) ==
        unfolding X(x) in
            unfolding i32(x.f_X_a) in x.f_X_a.int_val
```

Figure 5.1.: The snapshot type encoding for `struct X { a: i32 }`. Some details, such as an injectivity axiom for the conversion, have been omitted here for simplicity.

returning `true` *iff* the postcondition holds for the given old/new instances, arguments, and the result. This does not necessarily imply that the precondition held for the old arguments; use `pre$T()` for that.

Immutable arguments (`i32`, `&U`, …) need not be passed twice to `post$T()`, because their old and new value would always be the same. For example, for an argument `i` of type `i32`, `post$T()` would receive only one argument `i: Int` (instead of `oldi: Int` and `i: Int`).

- `hist_inv$T()`, taking two closure instances (of the same type) and checking whether the history invariant holds between them (the first argument is the old instance). This also checks all single-state invariants for the two argument instances; in particular, to check whether the single-state invariants hold for a given instance `cl`, one can call `hist_inv$T(cl, cl)`. Note that this means that `hist_inv$T(a, b)` is not a reflexive relation: `hist_inv$T(a, a)` does not hold for all `a`, only the ones that fulfill the single-state invariants. `hist_inv$T()` *is* transitive, though; and we can even give this weakened version of reflexivity (which is also used by Cohen *et al.* [3] to express that two-state invariants and `old()` "should […] only [be] used to describe how objects change, and not what […] their proper values [are]"), which we refer to as *pseudo*-reflexivity:

$$\forall f : T.\ \left((\exists f' : T.\ \texttt{hist\_inv\$T}\left(f', f\right)\right) \implies \texttt{hist\_inv\$T}\left(f, f\right)\right)$$

```rust
let x = 42;
let cl_add = |y: i32| -> i32 { x + y };

let mut count = 0;
let mut inc = || -> i32 { let r = count; count += 1; r };

let mut bx = Box::new (42);
let mut cl_drop = || -> i32 { let r = *bx; std::mem::drop(bx); r };
```

Figure 5.2.: Three examples of Rust closures, one for each of the `Fn`, `FnMut`, and `FnOnce` traits.

or, equivalently [4, Section "Transformation Rules for Implication"]:

$$\forall f, f' : T.\ \left(\texttt{hist\_inv\$T}\left(f', f\right) \implies \texttt{hist\_inv\$T}\left(f, f\right)\right)$$

This rule gives us reflexivity of the history invariants modulo the single-state invariants. For instance, after a call to `f`, we can inhale `hist_inv$T(old(f), f)` (because calls preserve the invariants), which then allows us to deduce `hist_inv$T(f, f)`, i.e. that the single-state invariants hold for the new instance.

5.1.1. Basics

Proving the Closure Code

Proving closure specifications works the same way as proving regular functions: Inhale the precondition, insert the closure code, exhale the postcondition. In fact, in Rust's MIR (Mid-level Intermediate Representation [11]), which serves as the basis for the Viper encoding in Prusti, closures are encoded as regular functions (with an extra argument for the captured state).

This also causes some restrictions, as already discussed in Section 4.1.2; in particular, all knowledge about the enclosing scope, such as the values of immutable captures, is lost and must be encoded somewhat more verbosely through views, to allow the instantiation-site to reason about the closure's captured state using its knowledge about the captured variables at the time of the capture. On the plus side, though, this leads to greater modularity and performance, because encoding closures as separate functions means they can be verified independently, even in parallel, and/or cached, and because the bodies of the Rust functions containing the closure definition are made smaller this way, their verification should also become faster and more easily manageable for the SMT solver (as opposed to inlining the closure bodies into the context of the containing function).

To illustrate how closures are encoded in the MIR, consider the three examples in Figure 5.2; conceptually, the translation to MIR will generate the code shown in Figure 5.3. In other words, there will be a fresh aggregate ("struct-like") type for every closure definition, with fields for every captured variable, taking into account the type of the captured variable and the capture kind (by immutable or mutable reference or by value), as well as a function pointer field that points to the closure code (not shown here, because it is implicit in the type, but every `ClAdd` object, for instance, would point to the `cl_add`

```rust
struct ClAdd<'x> {
    x: &'x i32
}

fn cl_add(cs: &ClAdd, y: i32) -> i32 {
    *cs.x + y
}

struct Inc<'count> {
    count: &'count mut i32
}

fn inc(cs: &mut Inc) -> i32 {
    let r = *cs.count;
    *cs.count += 1;
    r
}

struct ClDrop {
    bx: Box<i32>
}

fn cl_drop(cs: ClDrop) -> i32 {
    let r = *cs.bx;
    std::mem::drop(cs.bx);
    r
}
```

Figure 5.3.: The MIR encoding of the three examples from Figure 5.2, written in standard Rust, for legibility.

function).

As explained above, encoding the closure bodies works just as for regular functions. The body may assume `pre$T(self, ...) && hist_inv$T(self, self)`, i.e. it may assume the precondition and the invariants on the captured state. At the end of the encoding of the body, we must be able to prove `post$T(oldself, self, ...) && hist_inv$T(oldself, self)`, i.e. that the postcondition holds and the invariants have been preserved. Additionally, the relevant permisions for `self`, the arguments, and the result need to be encoded, as given by the signature (again, as for regular functions).

As an example, recall the `cl_add` closure from above, this time with a simple specification:

```
let x = 42;
let cl_add =
    // view: x: i32 = x
    // ensures: result == y + x
    |y: i32| -> i32 { x + y };
```

The body of `cl_add` would be encoded as shown in Figure 5.4. Note how the precondition specification function references the view `x` of `cl_add` by calling the pure `ClAdd$view$x()` function, which encodes the view, with the snapshot instance; the `to_snap$ClAdd()` conversion function establishes the correspondence between the view and the relevant field on the heap. Additionally, the closure guarantees that the old and new instance values are equal, because the body only receives a partial permission to `self`; this is because `cl_add` implements the `Fn` trait and thus receives only an immutable `self` reference; `FnMut` and `FnOnce` closures would need a full permission here, and `FnOnce` closures would not give back any permissions to `self`.

Checking Well-Formedness of Invariants

Checking well-formedness of invariants needs to happen somewhere, but it does not matter whether this is done as part of the encoding of the closure body or elsewhere, e.g. in a separate method.

Recall that all invariants must be pseudo-reflexive and transitive. Assume that the closure type will be named `T`; then we can check pseudo-reflexivity and transitivity by quantifying over the snapshot type for `T` (no triggers are necessary here, because we are only interested in proving these assertions):

```
assert forall cs: T :: (exists cs_: T :: hist_inv$T(cs_, cs))
    ==> hist_inv$T(cs, cs)

assert forall cs1: T, cs2: T, cs3: T ::
    (hist_inv$T(cs1, cs2) && hist_inv$T(cs2, cs3))
        ==> hist_inv$T(cs1, cs3)
```

We cannot encode these properties as axioms, because this would interfere with actually checking these properties (they would be trivially satisfied by those axioms in that case). Therefore, every Rust function that works with the closure type `T` has to inhale those properties in its body.

Encoding Views

Views are pure functions with a special argument, the closure instance, similar to how closures receive an implicit "self" argument. We can encode them as follows, for instance for the `inc` example already encountered above:

```
// Define snapshot type and specification functions for cl_add
domain ClAdd {}
function ClAdd$view$x(cl: ClAdd): Int
function pre$ClAdd(cl: ClAdd, i: Int): Bool { true }
function post$ClAdd(oldcl: ClAdd, cl: ClAdd, i: Int, r: Int): Bool {
    r == i + ClAdd$view$x(cl)
}
function hist_inv$ClAdd(oldcl: ClAdd, cl: ClAdd): Bool { true }

// The captured variable, encoded as Int for simplicity
field cl_capture$ClAdd$x: Int

predicate ClAdd$pred(r: Ref) {
    acc(r.cl_capture$ClAdd$x)
}

function read(): Perm
    ensures result > 0/1 && result < 1/1

// Conversion function from reference to snapshot type
function to_snap$ClAdd(r: Ref): ClAdd
    requires acc(ClAdd$pred(r), read())
    ensures ClAdd$view$x(result) ==
        unfolding acc(ClAdd$pred(r), read()) in r.cl_capture$ClAdd$x

// The actual encoding of the body
method encode_body$ClAdd (self: Ref, y: Int) returns (r: Int)
    requires acc(ClAdd$pred(self), read())
    requires let ss == (to_snap$ClAdd(self)) in
        pre$ClAdd(ss, y) && hist_inv$ClAdd(ss, ss)
    ensures acc(ClAdd$pred(self), read())
    ensures let oldss == (old(to_snap$ClAdd(self))) in
        let ss == (to_snap$ClAdd(self)) in
            post$ClAdd(oldss, ss, y, r)
            && hist_inv$ClAdd(oldss, ss)
{
    unfold acc(ClAdd$pred(self), read())
    r := self.cl_capture$ClAdd$x + y
    fold acc(ClAdd$pred(self), read())
}
```

Figure 5.4.: Encoding of the `cl_add` closure's body.

```
let mut count = 0;
let mut inc =
    // view: cnt: i32 = count
    || -> i32 { let r = count; count += 1; r };
```

This would be encoded in Viper as follows:

```
domain Inc {}
function inc$view$cnt(cl: Inc): Int

field cl_capture$inc$count: Int

predicate inc$pred(r: Ref) {
    acc(r.cl_capture$inc$count)
}

function read(): Perm
    ensures result > 0/1 && result < 1/1

function to_snap$inc(r: Ref): Inc
    requires acc(inc$pred(r), read())
    ensures inc$view$cnt(result) ==
        unfolding acc(inc$pred(r), read()) in
            r.cl_capture$inc$count
```

Note how the `to_snap$inc()` function, which converts a reference to the snapshot type, creates the correspondence between the view and its definition—in other words, between the view function `inc$view$cnt()` and the value of the `cl_capture$inc$count` field.

The same technique can be applied for views with extra arguments; consider this example:

```
let mut nums = vec! [1, 2, 3];
let cl =
    // view: el: (idx: usize) -> Option<i32> = nums.get(idx)
    // ensures: result == el(i)
    |i: usize| { nums.get(i) };
```

The view could conceptually be encoded as follows (we show the roughly corresponding Rust code here, for simplicity, to avoid having to encode `Vec` and `Option` in Viper):

```
#[pure]
fn cl_view_el (cs: &Cl, idx: usize) -> Option<i32> {
    cs.nums.get (idx)
}
```

Proving Closure Instantiations

Closure instantiation conceptually works just like struct instantiation; for instance, the instantiations of the three examples from Figure 5.2 could be encoded as follows:

```
let x = 42;
let cl_add = ClAdd { x: &x };
```

```
let mut count = 0;
let mut inc = Inc { count: &mut count };

let mut bx = Box::new (42);
let mut cl_drop = ClDrop { bx: bx };
```

Additionally, we need to prove that every newly created instance establishes its invariants. This can be achieved using the `hist_inv$T()` specification function, thanks to the (pseudo-)reflexivity requirement on history invariants: Asserting `hist_inv$T(cl, cl)` for some newly-created closure instance `cl` will check that all invariants have been established; in particular, the single-state invariants, because all two-state invariants must hold trivially (otherwise they would not be well-formed, i.e. reflexive).

Proving Closure Calls

Encoding closure calls is very simple, thanks to the specification functions: We only have to assert `pre$T(cl, a1, a2, ...)` (where `cl` is the closure instance of type `T` and `a1`, `a2`, *etc.* are the arguments to the call, all as snapshots) and `hist_inv$T(cl, cl)` (to make sure the instance is valid, i.e. satisfies all single-state invariants), exhale the relevant permissions (as for regular function calls; this is necessary for havocing memory locations that could be mutated by the closure), inhale the resulting permissions (again as for regular function calls), and assume `post$T(old_cl, cl, old_a1, a1, ...)`. Additionally, we can assume `hist_inv$T(old_cl, cl)`, because history invariants are always preserved by calls to the closure (and that allows us to infer `hist_inv$T(cl, cl)` for the new instance `cl`, using the pseudo-reflexivity requirement on `hist_inv$T()` discussed earlier).

5.1.2. Specification Entailments

Basic Idea

The main advantage of the specification function approach is that it allows us to delegate reasoning about closure specifications to the SMT solver, so that we can take advantage of the full capabilities of the SMT solver for instantiating/proving entailments under conjunctions, quantifiers, implications, *etc.* Consider this example:

```
let cl =
    // requires: a >= -10
    // ensures: result == a
    |a: i32| a + 10;
```

Now assume we want to prove `cl |= |a| { requires: a >= 0, ensures: result >= 0}`. This strengthens the precondition and weakens the postcondition (assuming the strengthened precondition, as explained in Section 2.3.2). The basic idea for encoding this in Viper is shown in Figure 5.5 (the assertions could also be condensed into a single statement). We can thereby leave it up to the SMT solver to figure out the precise instantiations of the quantifiers. `CL` refers to the snapshot type of the closure type. Note that whenever we are reasoning about a closure's pre- or postcondition (e.g. when encoding a call, or another entailment), we will refer to the `pre$CL()`/`post$CL()` specification functions, and thus the triggers provided will be sufficient.

```
// This is what we get from the closure definition
function pre$CL (cl: CL, a: Int): Bool {
    a >= -10
}
function post$CL (oldcl: CL, cl: CL, a: Int, res: Int): Bool {
    res == a
}

// This represents the entailed specification we want to prove
assert forall cl: CL, a: Int :: {pre$CL(cl, a)}
    a >= 0 ==> pre$CL(cl, a)
assert forall oldcl: CL, cl: CL, a: Int, res: Int ::
    {post$CL(oldcl, cl, a, res)}
    a >= 0 ==> (post$CL(oldcl, cl, a, res) ==> res >= 0)
```

Figure 5.5.: Preliminary encoding of a specification entailment, demonstrating the basic idea but neglecting, for the moment, the invariants.

In particular, this approach will work nicely for instantiating multiple entailments at once, as discussed in Figure 2.1. Here is a simplified version:

```
// assume cl |= |i| { requires: i >= 0 }
// assume cl |= |i| { requires: i < 0 }
// assert cl |= |i| { requires: true }
```

which could be encoded as follows:

```
assume forall cl: CL, i: Int :: {pre$CL(cl, i)}
    i >= 0 ==> pre$CL(cl, i)
assume forall cl: CL, i: Int :: {pre$CL(cl, i)}
    i < 0 ==> pre$CL(cl, i)

assert forall cl: CL, i: Int :: {pre$CL(cl, i)}
    true ==> pre$CL(cl, i)
```

In other words, all of the applications discussed in Section 4.2.2 are natively supported without any extra effort on our part by pushing the reasoning down to the SMT solver. The nested entailments from Section 4.2.4 can also be straightforwardly encoded using nested quantifiers, without the need for special handling.

Encoding `outer()` doesn't need any special attention, because it is mostly just a disambiguation device, anyway; for instance, `cl |= |x: i32| { requires: x >= outer(*y) }` is encoded as follows:

```
forall cl: CL, x: Int :: {pre$CL(cl, x)}
    x >= y.int_val ==> pre$CL(cl, x)
```

And finally, entailments across multiple calls can be handled by quantifying over multiple instances and sets of arguments, making sure that all the respective preconditions are implied, *etc.* For instance,

say we want to encode an antisymmetry check for a comparator function on integers (`cl |= |a, b|, |c, d| { ensures: a == d && b == c && result0 && result1 ==> a == b }`):

```
forall cl1: CL, cl2: CL, a: Int, b: Int, c: Int, d: Int ::
    {pre$CL(cl1, a, b), pre$CL(cl2, c, d)}
    true ==> (pre$CL(cl1, a, b) && (pre$CL(cl2, c, d)))

forall oldcl1: CL, cl1: CL, oldcl2: CL, cl2: CL,
    a: Int, b: Int, c: Int, d: Int, res0: Bool, res1: Bool ::
    {post$CL(oldcl1, cl1, a, b, res0),
     post$CL(oldcl2, cl2, c, d, res1)}
    true ==>
      (post$CL(oldcl1, cl1, a, b, res0)
       && post$CL(oldcl2, cl2, c, d, res1)) ==>
          (a == c && b == d && res0 && res1 ==> a == b)
```

Introducing Invariants

We now want to restrict the quantification over closure instances to *reachable* instances, meaning only those instances that fulfill the single-state invariants and the history invariants with regards to the current instance value. For instance, assuming the current closure instance is called `cl` and its snapshot type is `CL`, we can encode `cl |= |i| { requires: i >= 0, ensures: result >= 0 }` as follows:

```
assert forall cl_: CL, i: Int :: {pre$CL(cl_, i)}
    hist_inv$CL(cl, cl_) ==>
        (i >= 0 ==> pre$CL(cl_, i))
assert forall oldcl: CL, newcl: CL, i: Int, res: Int ::
    {post$CL(oldcl, newcl, i ,res)}
    hist_inv$CL(cl, oldcl) ==>
        (i >= 0 ==>
            ((post$CL(oldcl, newcl, i, res)
              && hist_inv$CL(oldcl, newcl)) ==> res >= 0))
```

i.e., we only talk about "future" instances, as permitted by the invariants. This is the final rule for encoding specification entailments. Note how `hist_inv$CL()` occurs twice in the postcondition encoding: This is necessary because `post$CL()` does not have to imply anything about the invariants; of course, the closure *body* does guarantee preservation of the invariants, but in our approach, invariants are handled separately in the `hist_inv$CL()` function. Here is a short example which demonstrates the problem:

```
let mut x = 0;
let mut cl =
    // view: x: i32 = x
    // invariant: x % 10 == 0
    // ensures: result == x
    || -> i32 { x += 10; x };

// assert cl |= || { ensures: result % 2 == 0 }
```

```
domain CL {}
function CL$view$x(cl: CL): Int
function pre$CL(cl: CL): Bool { true }
function post$CL(oldcl: CL, cl: CL, r: Int): Bool {
    r == CL$view$x(cl)
}
function hist_inv$CL(oldcl: CL, cl: CL): Bool {
    CL$view$x(oldcl) % 10 == 0
    && CL$view$x(cl) % 10 == 0
}

method test ()
{
    var cl: CL
    assume CL$view$x(cl) == 0

    // Fails:
    // assert forall oldcl_: CL, cl_: CL, r: Int ::
    //     hist_inv$CL(cl, oldcl_) ==> (true ==>
    //         (post$CL(oldcl_, cl_, r) ==> r % 2 == 0))

    assert forall oldcl_: CL, cl_: CL, r: Int ::
        hist_inv$CL(cl, oldcl_) ==> (true ==>
            ((post$CL(oldcl_, cl_, r)
              && hist_inv$CL(oldcl_, cl_)) ==> r % 2 == 0))
}
```

Figure 5.6.: Encoding of a specification entailment that relies on the invariant being established in the poststate (heap encoding and precondition entailment omitted for simplicity).

In order to prove the postcondition in the specification entailment, we need to know that the invariant holds for the captured state in the poststate. The Viper encoding is shown in Figure 5.6.

Encoding `|=!`

To encode specification entailments for single calls, expressed via the `|=!` operator, we can simply use the current closure instance (instead of quantifying over instance values). For instance, assume we want to prove an entailment `cl |=! |i: i32| { requires: i > 0, ensures: result > 0 }` with the snapshot type for `cl` being called `CL`:

```
assert forall i: Int :: {pre$CL(cl, i)}
    i > 0 ==> pre$CL(cl, i)
assert forall cl_: CL, i: Int, res: Int ::
    {post$CL(cl, cl_, i, res)}
    i > 0 ==> ((post$CL(cl, cl_, i, res)
               && hist_inv$CL(cl, cl_)) ==> res > 0)
```

Here, we don't quantify over the prestate instance `cl`, which is taken from the current state; we do, however, quantify over the arguments, the poststate instance `cl_` (which isn't known yet), and the result.

Invariants in Specification Entailments

Finally, entailing invariants works by quantifying over two instances at once; for instance, we can encode `cl |= || { invariant: cnt(self) >= old(cnt(self)) }` as

```
exhale forall oldcl: CL, newcl: CL ::
    {hist_inv$CL(oldcl, newcl)}
    hist_inv$CL(oldcl, newcl) ==> (cnt(newcl) >= cnt(oldcl))
```

Note that we have to encode well-formedness checks for the weakened invariant separately; an argument for why neither pseudo-reflexivity nor transitivity are necessarily preserved when weakening invariants is given in Appendix A.

A Complete Example

To illustrate how the previously discussed concepts and encoding strategies fit together, consider this Rust example:

```
let mut count = 0;
let inc =
    // view: cnt: i32 = count
    // invariant: old(cnt) <= cnt
    // ensures: result == old(cnt) && cnt == old(cnt) + 1
    move || -> i32 { let r = count; count += 1; r };

// assert inc |=! || { ensures: result == 0 }
let x = inc ();
assert! (x == 0);

// assert inc |= || { ensures: result >= 1 }
```

The following Viper code, which has been successfully verified (as all the other Viper examples in this section), demonstrates the encoding of the example above, using the techniques just described. The heap encoding is slightly simplified: Integers are encoded as just `Int`.

```
1  // Snapshot type and specification functions for the closure type
2  domain Inc {}
3
4  function pre$inc (inc: Inc): Bool {
5      true
6  }
7
8  function post$inc (old_inc: Inc, inc: Inc, res: Int): Bool {
9      (res == inc$view$cnt (old_inc))
10     && (inc$view$cnt (inc) == inc$view$cnt (old_inc) + 1)
11 }
```

```
12
13 function hist_inv$inc (old_inc: Inc, inc: Inc): Bool {
14     inc$view$cnt (old_inc) <= inc$view$cnt (inc)
15 }
16
17 function inc$view$cnt (inc: Inc): Int
18
19
20 // Heap encoding and snapshot conversion function
21 field cl_capture$inc$count: Int
22
23 predicate inc$pred (r: Ref) {
24     acc (r.cl_capture$inc$count)
25 }
26
27 function read (): Perm
28     ensures result > 0/1 && result < 1/1
29
30 function inc$to_snapshot (inc: Ref): Inc
31     requires acc (inc$pred(inc), read())
32     ensures inc$view$cnt(result) ==
33         (unfolding acc(inc$pred(inc), read()) in
34             inc.cl_capture$inc$count)
35
36
37 // Encoding of closure body
38 method encode_inc_body (self: Ref) returns (res: Int)
39     requires acc(inc$pred(self))
40     requires let self_snap == (inc$to_snapshot(self)) in
41         hist_inv$inc(self_snap, self_snap)
42         && pre$inc(self_snap)
43     ensures acc(inc$pred(self))
44     ensures let self_snap == (inc$to_snapshot(self)) in
45         let old_self_snap == (old(inc$to_snapshot(self))) in
46             hist_inv$inc(old_self_snap, self_snap)
47             && post$inc(old_self_snap, self_snap, res)
48 {
49     {
50         // Check well-formedness of the invariants; these
51         // quantifiers do not need triggers, because we only
52         // prove but never use them (in this method)
53         assert forall cl: Inc ::
54             (exists cl_: Inc :: hist_inv$inc(cl_, cl))
55             ==> hist_inv$inc (cl, cl)
56         assert forall cl1: Inc, cl2: Inc, cl3: Inc ::
57             hist_inv$inc (cl1, cl2) && hist_inv$inc (cl2, cl3)
58                 ==> hist_inv$inc (cl1, cl3)
59     }
60
```

```
61      // Encode body
62      unfold inc$pred(self)
63      var r: Int := self.cl_capture$inc$count
64      self.cl_capture$inc$count := self.cl_capture$inc$count + 1
65      res := r
66      fold inc$pred(self)
67  }
68
69  // Encodes the enclosing function, in principle, but details
70  // omitted here
71  method encode_test ()
72  {
73      var count: Int := 0
74
75      // Encode well-formedness property of the invariants
76      assume forall cl: Inc :: {hist_inv$inc(cl, cl)}
77          (exists cl_: Inc :: hist_inv$inc(cl_, cl))
78          ==> hist_inv$inc(cl, cl)
79      assume forall cl1: Inc, cl2: Inc, cl3: Inc ::
80          {hist_inv$inc(cl1, cl2), hist_inv$inc(cl1, cl3)}
81          {hist_inv$inc(cl2, cl3), hist_inv$inc(cl1, cl3)}
82          (hist_inv$inc(cl1, cl2) && hist_inv$inc(cl2, cl3))
83          ==> hist_inv$inc(cl1, cl3)
84
85      // Encode instance creation
86      var inc: Ref
87      inhale acc (inc$pred(inc))
88          && unfolding inc$pred(inc) in
89              inc.cl_capture$inc$count == count
90
91      // Encode |=! specification entailment
92      var inc_snap: Inc := inc$to_snapshot (inc)
93      exhale true ==> pre$inc (inc_snap)
94          && forall new_inc: Inc, res: Int ::
95                 {post$inc(inc_snap, new_inc, res)}
96                 true ==> ((post$inc (inc_snap, new_inc, res)
97                           && hist_inv$inc (inc_snap, new_inc))
98                          ==> res == 0)
99
100     // Encode call
101     var inc_pre_snap: Inc := inc$to_snapshot (inc)
102     assert pre$inc (inc_pre_snap)
103         && hist_inv$inc(inc_pre_snap, inc_pre_snap)
104     exhale acc (inc$pred(inc))
105     inhale acc (inc$pred(inc))
106     var x: Int
107     var inc_post_snap: Inc := inc$to_snapshot (inc)
108     assume post$inc (inc_pre_snap, inc_post_snap, x)
109         && hist_inv$inc(inc_pre_snap, inc_post_snap)
```

```
110
111     // Encode assertion
112     assert x == 0
113
114     // Encode specification entailment
115     inc_snap := inc$to_snapshot (inc)
116     exhale (forall cl: Inc :: {pre$inc(cl)}
117             hist_inv$inc(inc_snap, cl)
118                 ==> (true ==> pre$inc (cl)))
119         && (forall old_cl: Inc, new_cl: Inc, res: Int ::
120                 {post$inc(old_cl, new_cl, res)}
121                 hist_inv$inc (inc_snap, old_cl) ==>
122                     (true ==> ((post$inc (old_cl, new_cl, res)
123                                 && hist_inv$inc(old_cl, new_cl))
124                                    ==> (res >= 1))))
125 }
```

This concludes our discussion of the encoding of specification entailments. Typically, specification entailments will occur in the specifications of higher-order functions; thus, we will look at how to encode them next.

5.1.3. Higher-Order Functions and Boxed Closures

We have described above how specification functions are defined for every individual closure type (such as `pre$T()` for closure type `T`). This is necessary because the captured state, views, and history invariants may differ between different closure types. However, this poses a problem for generic higher-order functions with closure type arguments (such as `fn foo<T: FnMut () -> i32> (...)`), because the concrete type will not be known at the definition site, and in fact the same generic function can be called from multiple locations with different type arguments.

We propose two methods for solving this problem: Rewriting the specifications and using "abstract" specification functions. We will discuss both of them here, to discuss their respective advantages and drawbacks.

Rewriting Specifications

To verify a generic higher-order function modularly, we may not make any assumptions about its type arguments (beyond any type bounds, as given e.g. in `where` clauses). Therefore, in order to verify the higher-order function, we define a fresh closure snapshot type `F`, along with a set of specification functions (`pre$F()` *etc.*), for which we do not provide bodies. Then, we can encode the pre- and postcondition (meaning, in particular, the specification entailments and arrows they contain) in terms of these specification functions.

At the call-site of the higher-order function, we *do* know the concrete type arguments.[3] This means that when encoding the pre- and postcondition at the call-site (asserting the former and assuming the

[3] In fact, Rust monomorphizes generic functions, meaning calls are always to a specific instantiation, separate from all other instantiations, of a generic function.

latter), we can perform this encoding in terms of the concrete specification functions (`pre$T()` *etc.*) for the concrete type `T`. Note that if one generic higher-order function calls another, the "concrete" type `T` would again be a fresh type, used in the encoding of *that* higher-order function.

For illustration, consider the following example:

```
// requires: cl |= |i| { requires: i == 42, ensures: result >= 0 }
// ensures: result >= 0
fn hof<F: FnMut (i32) -> i32> (mut cl: F) -> i32 {
    cl (42)
}

let cl =
    // ensures: result == i
    |i| i;
let r = hof (cl);
assert! (r >= 0);
```

Using the specification rewriting approach, this would be encoded as follows; note how the specification used for `hof()` at its definition-site talks about different specification functions than the call to `hof()`, which is implicit in the `example()` method by asserting the pre- and assuming the postcondition. Details of the heap encoding, the closure body proof, and the pseudo-reflexivity and transitivity properties of the history invariants are omitted here to focus the example on the higher-order function encoding:

```
1  // Fresh snapshot type and specification functions for hof's
2  // generic type parameter F
3  domain FCL {}
4  function pre$fcl(cl: FCL, i: Int): Bool
5  function post$fcl(oldcl: FCL, cl: FCL, i: Int, res: Int): Bool
6  function hist_inv$fcl(oldcl: FCL, cl: FCL): Bool
7
8  // Encoding of the higher-order function hof
9  method hof (cl: FCL) returns (r: Int)
10     requires hist_inv$fcl(cl, cl)
11     requires forall cl_: FCL, i: Int :: {pre$fcl(cl_, i)}
12         hist_inv$fcl(cl, cl_) ==> (i == 42 ==> pre$fcl(cl_, i))
13     requires forall oldcl_: FCL, cl_: FCL, i: Int, res: Int ::
14         {post$fcl(oldcl_, cl_, i, res)}
15         hist_inv$fcl(cl, oldcl_) ==> (i == 42 ==>
16             ((post$fcl(oldcl_, cl_, i, res)
17               && hist_inv$fcl(oldcl_, cl_)) ==> res >= 0))
18     ensures r >= 0
19 {
20     assert pre$fcl(cl, 42) && hist_inv$fcl(cl, cl)
21     var newcl: FCL
22     var res: Int
23     assume post$fcl(cl, newcl, 42, res)
24         && hist_inv$fcl(cl, newcl)
25     r := res
26 }
```

```
27
28
29  // Snapshot type and specification functions for the concrete
30  // closure type used in the example at hof's call-site
31  domain CL {}
32  function pre$cl(cl: CL, i: Int): Bool { true }
33  function post$cl(oldcl: CL, cl: CL, i: Int, res: Int): Bool {
34      res == i
35  }
36  function hist_inv$cl(oldcl: CL, cl: CL): Bool { true }
37
38  // Encoding of hof's call-site
39  method example ()
40  {
41      var cl: CL
42      var r: Int
43      assert hist_inv$cl(cl, cl)
44      assert forall cl_: CL, i: Int :: {pre$cl(cl_, i)}
45          hist_inv$cl(cl, cl_) ==> (i == 42 ==> pre$cl(cl_, i))
46      assert forall oldcl_: CL, cl_: CL, i: Int, res: Int ::
47          {post$cl(oldcl_, cl_, i, res)}
48          hist_inv$cl(cl, oldcl_) ==> (i == 42 ==>
49              ((post$cl(oldcl_, cl_, i, res)
50                && hist_inv$cl(oldcl_, cl_)) ==> res >= 0))
51      assume r >= 0
52
53      assert r >= 0
54  }
```

Here, we define two snapshot types (and, accordingly, two sets of specification functions): First, the `FCL` type, which we use throughout the higher-order function's encoding. The `FCL` type represents the generic argument to the higher-order function; thus, we may not make any assumptions about it and provide specification functions without bodies.

Second, the `CL` type is used to encode the concrete closure type used at the higher-order function's call-site. The specification functions for `CL` have bodies, which reflect the (known) specification for the `cl` closure. When encoding the higher-order function call to `hof`, we have to rewrite `hof`'s pre- and postcondition to substitute the calls to `FCL`'s specification functions with the respective calls to `CL`'s specification functions.

The advantage of this approach is that it is relatively simple to implement, and that we can readily use all of the knowledge available at the higher-order function's call-site about the concrete closure's (`CL`'s in the example) specification functions in order to prove the higher-order function's precondition.

The downside is that this approach only works as long as we have a concrete type (which could be another generic type, as noted above) to rewrite the higher-order function's specification for. In particular, this means that boxed closures are not supported by this approach: Boxes containing a dynamic type (`Box<dyn Fn...>`) can point to different concrete types at different program points; thus, there is no single type we can rewrite the specification for. The next section presents a different

approach better suited to this application.

Abstract Specification Functions

The second possibility for encoding higher-order functions and their specifications is to globally define one more set of specification functions (`pre$S()` *etc.*) per closure *signature* S (such as `(i32, i32) -> i32`), along with a fresh snapshot type for S. Then, any generic higher-order function that does not know the concrete types of its argument closures could be encoded in terms of these "abstract" (because they do not belong to any concrete closure type) specification functions.

At the call-site, then, we could reuse the same specification without the need for any rewriting; instead, at the definition-site of every closure, we need to add information about the correspondence between the concrete (`pre$T()` *etc.* for the concrete closure type `T`) and abstract (`pre$S()` *etc.* for the signature S) specification functions.

Here is the same example from the previous section again, this time encoded using abstract specification functions, and without rewriting any specifications. Note how the abstract snapshot type is used in the encoding of the higher-order function *and* at its call-site, and how the correspondence axioms `{pre,post,hist_inv}_corr` establish the relation between the specification functions of the concrete closure type and the abstract ones:

```
1  // Snapshot type and specification functions for signature |i32| -> i32
2  domain S {
3      function pre$S(cl: S, i: Int): Bool
4      function post$S(oldcl: S, cl: S, i: Int, res: Int): Bool
5      function hist_inv$S(oldcl: S, cl: S): Bool
6  }
7
8  // Encoding of the higher-order function hof
9  method hof (cl: S) returns (r: Int)
10     requires hist_inv$S(cl, cl)
11     requires forall cl_: S, i: Int :: {pre$S(cl_, i)}
12         hist_inv$S(cl, cl_) ==> (i == 42 ==> pre$S(cl_, i))
13     requires forall oldcl_: S, cl_: S, i: Int, res: Int ::
14         {post$S(oldcl_, cl_, i, res)}
15         hist_inv$S(cl, oldcl_) ==> (i == 42 ==>
16             ((post$S(oldcl_, cl_, i, res)
17               && hist_inv$S(oldcl_, cl_)) ==> res >= 0))
18     ensures r >= 0
19 {
20     assert pre$S(cl, 42) && hist_inv$S(cl, cl)
21     var newcl: S
22     var res: Int
23     assume post$S(cl, newcl, 42, res) && hist_inv$S(cl, newcl)
24     r := res
25 }
26
27
28 // Snapshot type and specification functions for the concrete
29 // closure type, with correspondence axioms for the abstract
```

```
// specification functions
domain CL {
    function pre$cl(cl: CL, i: Int): Bool
    axiom { forall cl: CL, i: Int :: {pre$cl(cl, i)}
        pre$cl(cl, i) <==> true
    }
    function post$cl(oldcl: CL, cl: CL, i: Int, res: Int): Bool
    axiom { forall oldcl: CL, cl: CL, i: Int, res: Int ::
        {post$cl(oldcl, cl, i, res)}
        post$cl(oldcl, cl, i, res) <==> res == i
    }
    function hist_inv$cl(oldcl: CL, cl: CL): Bool
    axiom { forall oldcl: CL, cl: CL :: {hist_inv$cl(oldcl, cl)}
        hist_inv$cl(oldcl, cl) <==> true
    }

    function upcast$CL$S(cl: CL): S
    function downcast$S$CL(s: S): CL

    axiom upcast_injective {
        forall cl: CL :: {upcast$CL$S(cl)}
            downcast$S$CL(upcast$CL$S(cl)) == cl
    }

    axiom pre_corr {
        forall cl: CL, i: Int, s: S ::
            {upcast$CL$S(cl), pre$S(s, i)}
            hist_inv$S(upcast$CL$S(cl), s) ==>
                (pre$S(s, i) <==> pre$cl(downcast$S$CL(s), i))
    }

    axiom post_corr {
        forall cl: CL, olds: S, s: S, i: Int, res: Int ::
            {upcast$CL$S(cl), post$S(olds, s, i, res)}
            hist_inv$S(upcast$CL$S(cl), olds) ==>
                (post$S(olds, s, i, res) <==>
                    post$cl(downcast$S$CL(olds),
                            downcast$S$CL(s), i, res))
    }

    axiom hist_inv_corr {
        forall oldcl: CL, cl: S ::
            {hist_inv$S(upcast$CL$S(oldcl), cl)}
            hist_inv$S(upcast$CL$S(oldcl), cl) <==>
                hist_inv$cl(oldcl, downcast$S$CL(cl))
    }
}

// Encoding of the higher-order function's call-site
```

```
79 method example ()
80 {
81     var cl: CL
82     var r: Int
83     r := hof(upcast$CL$S(cl))
84     assert r >= 0
85 }
```

Notice how we can simply call `hof()` in the body of the `example()` method, i.e. no rewriting takes place. Two main difficulties arise for this approach: First, the correspondence axioms need to relate not only the current instance and its upcasted value, but also all *future* instances that we can produce by encoding calls via the abstract specification functions (e.g. in a higher-order function) and then downcasting the resulting instances again. This is why choosing triggers for the correspondence axioms is not straightforward; we are not certain whether the choice made here is indeed sufficient in all cases.

Second, we should avoid downcasting an abstract instance of type `S` into `CL` unless we know that it actually *is* an instance of the concrete `CL` type; some further work may be necessary in this regard to encode this properly. Perhaps a kind of "discriminator" function that maps an instance of the abstract snapshot type to an integer denoting the concrete type could be used for this purpose (together with corresponding pre-/postconditions for the up-/downcasting functions).

Our implementation (cf. Section 5.2) uses the specification rewriting approach, but we have shown the abstract specification function approach here as well to demonstrate an alternative with different strengths and weaknesses; in particular, unlike the specification rewriting approach, abstract specification functions can be used when working with boxed closures, as discussed in the following section.

Boxed Closures

Boxed closures containing a `dyn Fn*` type (as well as dynamic references and function pointers) can point to instances of different closure/function types/definitions at different times; this means they pose a quite similar problem to generic higher-order functions, with one difference being that boxes can be reassigned to different instances (and thereby concrete types) at any point, whereas the closure type argument of a higher-order function is constant per instantiation/call.

Still, the approach using abstract specification functions described above works well for boxed closures: We can encode entailments involving a boxed closure with signature S in terms of the corresponding abstract specification functions (`pre$S()` *etc.*), just as before in the case of higher-order functions. For example:

```
// requires: **f |= || { ensures: result >= 0 }
// ensures: **f |= || { ensures: result < 0 }
fn foo (f: &mut Box<dyn FnMut () -> i32>) {
    f ();
    // assert hist_inv(old(**f), **f)
    *f = Box::new (
        // ensures: result == -42
        || -42);
}
```

Using abstract specification functions, this example would be encoded as follows (details of the heap encoding are completely omitted here, for simplicity; the poststate value of the argument reference is modeled as a return value). Notice how we only use one set of specification functions despite the fact that the concrete type stored in the box changes; we simply encode the new behavior in terms of the same abstract specification functions. This allows us to deal with the dynamic aspect of boxed closures (i.e. that the concrete type can change):

```
domain FCL {}
function pre$fcl (cl: FCL): Bool
function post$fcl (oldcl: FCL, cl: FCL, res: Int): Bool
function hist_inv$fcl (cl1: FCL, cl2: FCL): Bool

method havoc_fcl () returns (f: FCL)

method foo (f: FCL) returns (f_: FCL)
    requires hist_inv$fcl(f, f)
    requires forall nf: FCL :: {pre$fcl(nf)}
        hist_inv$fcl(f, nf) ==> (true ==> pre$fcl(nf))
    requires forall of: FCL, nf: FCL, res: Int ::
        {post$fcl(of, nf, res)}
        hist_inv$fcl(f, of) ==>
            (true ==> ((post$fcl(of, nf, res)
                        && hist_inv$fcl(of, nf)) ==> res >= 0))
    ensures hist_inv$fcl(f_, f_)
    ensures forall nf: FCL :: {pre$fcl(nf)}
        hist_inv$fcl(f_, nf) ==> (true ==> pre$fcl(nf))
    ensures forall of: FCL, nf: FCL, res: Int ::
        {post$fcl(of, nf, res)}
        hist_inv$fcl(f_, of) ==>
            (true ==> ((post$fcl(of, nf, res)
                        && hist_inv$fcl(of, nf)) ==> res < 0 ))
{
    assert pre$fcl(f) && hist_inv$fcl(f, f)
    f_ := havoc_fcl()
    var r: Int
    assume post$fcl(f, f_, r) && hist_inv$fcl(f, f_)

    // Encoding of closure body omitted; we model only the assignment here
    // by assuming the new specification (this would be derived from the
    // correspondence axioms on the concrete closure type in a complete
    // encoding):
    f_ := havoc_fcl()
    assume hist_inv$fcl(f_, f_)
    assume forall nf: FCL :: {pre$fcl(nf)}
        hist_inv$fcl(f_, nf) ==> (true ==> pre$fcl(nf))
    assume forall of: FCL, nf: FCL, res: Int ::
        {post$fcl(of, nf, res)}
        hist_inv$fcl(f_, of) ==> ((post$fcl(of, nf, res)
                                   && hist_inv$fcl(of, nf))
```

```
43                                          ==> res == -42)
44  }
```

Recall the framing issue from Section 4.2.7: `hist_inv(a, b)` for boxed closures `a` and `b` should only hold if `a` and `b` are instances of the same definition. This is achieved implicitly here, by inhaling `hist_inv$fcl(f, f_)` after the call but not after the assignment. Again, future work could improve on this, e.g. by using a discriminator function as mentioned above.

5.1.4. Arrow Notation

The arrow notation expresses that there has been a call such that certain assertions were true in the pre- and poststate. Consequently, we can easily encode this using existential quantifiers; for instance, take the following example:

```
// requires: f |= |i| { requires: i >= 0 }
// ensures: old(f) (:i) / { *i == 42 } ~~> { outer(result) == *i }
fn foo (f: impl FnMut (&mut i32) -> i32) -> i32 {
    let mut x = 42;
    f (&mut x);
    x
}
```

We can encode `foo`'s postcondition as follows (with details of the heap encoding omitted again, for simplicity):

```
method foo (f: F) returns (res: Int)
    ensures exists postf: F, i: Int, posti: Int, r: Int ::
        {pre$F(f, i), post$F(f, postf, i, posti, r)}
        pre$F(f, i) && post$F(f, postf, i, posti, r)
        && hist_inv$F(f, postf)
        && i == 42 && res == posti
```

Note how we do not quantify over `f`, because it is a known instance (namely, the one from `foo`'s prestate). We do quantify over the poststate instance value, the old/new arguments, and the result. The quantifier body expresses that the pre- and postcondition as well as the invariants hold for these values, by calling the respective specification functions, and encodes the pre- and poststate descriptions (the last two conjuncts above). Observe, in particular, how `*i` is encoded differently (`i` for encoding the prestate description, and `posti` for the poststate), and how `outer(result)` has been resolved to `res`, the result of the `foo()` method.

5.1.5. Ghost Arguments and Results

Prusti does not, at the time of writing, support ghost arguments/results; and to some extent, choices regarding their implementation are orthogonal to their application to the verification of closure code. Nonetheless, we can describe some general ideas here.

Note, first of all, that ghost arguments and results in specifications of higher-order functions do not actually have to be transferred between caller and callee; in a modular verification setting, both will be verified independently from each other, including the ghost arguments. This means that the callee will

have to be verified without any assumptions about the ghost arguments (except what is given by its precondition), and the caller has to check whether their concrete choice of ghost arguments satisfies the callee's precondition. Similarly, the caller can't assume anything about the ghost result except what is given by the callee's postcondition.

With this in mind, the callee can treat the ghost arguments as unknown constants, similar to an existential quantifier. The caller knows their values, because the responsibility for choosing them lies with the caller; thus, she may substitute the concrete values in the callee's specification and thereby proceed to prove the pre- and assume the postcondition. The reverse strategy can be applied to ghost results.

As for ghost argument/result functions, they can be encoded similarly: The callee defines a fresh function without a body in Viper, meaning this represents some unknown function with the given signature, and uses it to prove its (the callee's) body by substituting it for the ghost argument function. The caller defines a Viper function with a body, because she knows the ghost function's definition, and substitutes it in the callee's specification, or just substitutes the ghost function's definition directly in the higher-order function's specification. Then, she uses that to encode the function call.

To illustrate, consider this example:

```
// ghost_arg: cnt: (F) -> i32
// requires: f |= || { ensures: cnt(self) == old(cnt(self)) + 1 }
fn foo<F: FnMut () -> i32> (mut f: F) {
    // ...
}


let mut count = 0;
let mut cl =
    // view: c: i32 = count
    // invariant: c % 3 == 0
    // ensures: c == old(c) + 3
    || -> i32 { let r = count; count += 3; r };

// ghost_arg: cnt(f) := f.c / 3
foo (cl);
```

Here is a possible encoding in Viper, using the specification rewriting approach for the higher-order function call. Note how the higher-order function's specification is expressed in terms of the fresh bodyless `ghost_arg$foo$cnt()` function in `foo`'s precondition to make sure it will work for any concrete choice of the ghost argument. On the other hand, in the `example` method, when encoding the call to `foo`, the ghost argument is substituted with (the encoding of) its concrete definition (`f.c / 3`).

```
1 domain Inc {}
2 function inc$view$c(cl: Inc): Int
3 function pre$inc(cl: Inc): Bool { true }
4 function post$inc(oldcl: Inc, cl: Inc, res: Int): Bool {
5     inc$view$c(cl) == inc$view$c(oldcl) + 3
6 }
7 function hist_inv$inc(oldcl: Inc, cl: Inc): Bool {
```

```
8       inc$view$c(oldcl) % 3 == 0
9       && inc$view$c(cl) % 3 == 0
10  }
11
12  // Encoding of closure body omitted
13
14  method example ()
15  {
16      var cl: Inc
17      assume inc$view$c(cl) == 0
18      assert hist_inv$inc(cl, cl)
19
20      // Encode foo's precondition (i.e., the call to foo)
21      assert forall cl_: Inc :: {pre$inc(cl_)}
22          hist_inv$inc(cl, cl_) ==> (true ==> (pre$inc(cl_)))
23      assert forall oldcl_: Inc, cl_: Inc, r: Int ::
24          {post$inc(oldcl_, cl_, r)}
25          hist_inv$inc(cl, oldcl_) ==>
26              ((post$inc(oldcl_, cl_, r)
27                && hist_inv$inc(oldcl_, cl_)) ==>
28                  (inc$view$c(cl_) / 3) ==
29                  (inc$view$c(oldcl_) / 3) + 1)
30  }
31
32  domain F {}
33  function pre$f (cl: F): Bool
34  function post$f (oldcl: F, cl: F, res: Int): Bool
35  function hist_inv$f (oldcl: F, cl: F): Bool
36
37  function ghost_arg$foo$cnt (cl: F): Int
38
39  method foo (cl: F)
40      requires forall cl_: F :: {pre$f(cl_)}
41          hist_inv$f(cl, cl_) ==> (true ==> pre$f(cl_))
42      requires forall oldcl_: F, cl_: F, r: Int ::
43          {post$f(oldcl_, cl_, r)}
44          hist_inv$f(cl, oldcl_) ==>
45              ((post$f(oldcl_, cl_, r)
46                && hist_inv$f(oldcl_, cl_)) ==>
47                  ghost_arg$foo$cnt(cl_) ==
48                  ghost_arg$foo$cnt(oldcl_) + 1)
49  {
50      // ...
51  }
```

In this section, we have investigated strategies for the Viper encoding of the methodology from Chapter 4. All of the examples presented so far have been manually encoded, though; in the next section, we will look at how to automate this translation.

5.2. Implementation in Prusti

This section outlines some aspects of the current implementation, which automates the previously discussed encoding techniques as part of the Prusti verifier. Not all of the previously discussed features have been implemented yet; please refer to Section 6.2 for a discussion of several supported example programs, as well as of the current implementation's limitations.

Specification Parsing

Before thinking about the Viper encoding, we need to extend the Prusti parser to recognize and accept specification annotations for closures (and not only functions) and, as part of the specification language, specification entailments (and, later, the arrow notation, ghost arguments, *etc.*). This is achieved via so-called *procedural macros* [16, Section 19.5], which allow us to rewrite the abstract syntax tree of the input program after parsing. Closure specifications currently have the following syntax:

```
let cl = closure!(
    requires (i > 0),
    ensures (result >= i),
    |i: i32| -> i32 { i + 1 })
);
```

We must check the syntax of the pre- and postcondition expressions and make sure they type-check. For this purpose, we add dummy functions `prusti_pre()` and `prusti_post()`, with the same arguments as the closure (plus an extra `result` argument for `prusti_post`), and encode the pre- and postcondition expressions in their bodies, respectively:[4]

```
let cl = {
    |i: i32| -> i32 {
        if false {
            #[prusti::spec_only]
            fn prusti_pre(i: i32) {
                #[prusti::spec_only]
                || -> bool { i > 0 };
            }
        }

        let result = { i + 1 };

        if false {
            #[prusti::spec_only]
            fn prusti_post(i: i32, result: i32) {
                #[prusti::spec_only]
                || -> bool { result >= i };
```

[4] The fact that the expressions are encoded as bodies of closures within `prusti_{pre,post}` is an implementation detail motivated by how Prusti encodes assertions that transcend Rust syntax—for instance, an assertion `A ==> B` (consisting of Rust expressions `A` and `B` and the non-Rust implication operator) in the precondition would be encoded with two closures in `prusti_pre`'s body, one for encoding `A` and one for `B`. The two would be stitched back together in a later stage of Prusti.

```
            }
        }

        result
    }
};
```

Many additional annotations that Prusti uses to keep track of the specification (which expressions it consists of, *etc.*) are omitted here for legibility. The `prusti_pre()` and `prusti_post()` functions will now be type-checked by the compiler; for instance, if the precondition was not well-typed, then the Rust compiler would report a type error in `prusti_pre`'s body (specifically, the `|| -> bool { i > 0 }` helper closure, which could then be traced back by Prusti to the original `closure!()` macro expression the user wrote).

Encoding closure specifications as described is simple and similar to how Prusti encodes regular functions, thus allowing us to reuse parts of the existing code. However, it is not without drawbacks: Unlike functions, closure signatures may be *inferred*, i.e. the above example closure `cl` could also be defined as `|i| i + 1`. Because our rewriting step happens after parsing but before type-checking, we have to rely on explicit type annotations for closures in order to be able to generate the `prusti_pre()` and `prusti_post()` functions. Usually, this is at most an inconvenience for the user; the only serious restriction this causes is that higher-order closures are not supported by this approach, because giving explicit type annotations for them is not possible (as closures have anonymous types in Rust). This is a limitation that should be overcome in the future, given that supporting higher-order closures was one of our goals (our methodology and Viper encoding strategy already support higher-order closures). For the time being, note that higher-order *functions* are supported, which are more common in practice than higher-order closures.

Note that turning `prusti_pre` and `prusti_post` into closures (rather than functions, so that their argument types could be inferred as well) would not immediately solve this problem; assume we defined the closure as `|i| -> i32 { i }`, i.e. without type annotations for `i`. We might then be tempted to rewrite the code as follows:

```
let cl = {
    |i| -> i32 {
        if false {
            #[prusti::spec_only]
            let prusti_pre = |i| {
                #[prusti::spec_only]
                || -> bool { i > 0 };
            };
        }

        [...]
    }
};
```

Type inference for this fragment would fail, however, because we never use (e.g. call) the `prusti_pre` closure; thus, the compiler cannot infer its type. We could add a dummy call `prusti_pre(i);` after its definition, but if `i` turns out to be a non-copy type, this would move out `i`.

One more problem exists when rewriting `prusti_{pre,post}` as closures, even with type annotations; consider this slightly modified example:

```
let k: i32 = 42;
let cl = closure!(
    requires (k > 0),
    ensures (result >= i),
    |i: i32| -> i32 { i + 1 }
);
```

The precondition accesses some variable `k`, defined in the enclosing scope but neither shadowed by a closure argument nor captured by the closure. Now imagine we were to rewrite this example as follows:

```
let k: i32 = 42;
let cl = {
    |i: i32| -> i32 {
        if false {
            #[prusti::spec_only]
            let prusti_pre = |i: i32| {
                #[prusti::spec_only]
                || -> bool { k > 0 };
            };
        }

        [...]
    }
};
```

This would be accepted by the Rust compiler (`k` would simply be added to the list of variables captured by `prusti_pre` and `cl`), even though the precondition is illegal. Additional checks could circumvent this issue, but this would require non-trivial parsing logic in the implementation to handle shadowing, variable scopes, *etc.* properly.

Finally, note that without type annotations, the way a closure argument is used in the pre- or post-condition can influence type inference, which may also be undesirable, given that specifications are ghost code and should not influence the rest of the program. Future work should try to figure out an overall, comprehensive strategy for rewriting and type-checking closure specifications that prevents all of these issues.

Closure calls do not need any special syntax, i.e. the above closure could simply be called as, say, `cl(42)`. What we do still need to extend the parser for are specification entailments, e.g. as part of a higher-order function's specification. For instance, the following example is accepted by our parser:

```
#[requires(f |= |i: i32| [ requires(i >= 0), ensures(result >= 0) ])]
fn example<F: Fn (i32) -> i32> (f: F) { ... }
```

Again, some rewriting will take place to type-check the right-hand side of the entailment operator, with the aforementioned restrictions and difficulties.

Viper Encoding

Because closure bodies are encoded as regular functions in Rust's MIR, we can reuse Prusti's `ProcedureEncoder` to encode them as if they were regular functions. To encode closure calls, we need some special handling, because closures are not called directly in the MIR; for instance:

```
_4 = const <[closure@/tmp/cl.rs:5:13: 9:6]
    as std::ops::Fn<(i32, i32)>>
        ::call(move _5, move _6) -> [return: bb2, unwind: bb1];
```

Here, `[closure@/tmp/cl.rs:5:13: 9:6]` is a placeholder for the anonymous closure type, which is then upcasted to the `std::ops::Fn<(i32, i32)>` trait, on which the actual call happens by calling the `call` method. This method receives two arguments: The closure instance (`_5` in the example) and the arguments to the call, stored in a single tuple (`_6`).

Therefore, we can retrieve the actual closure definition that gets called through the type of `_5`; then, we can deconstruct the tuple `_6` to get the individual arguments to the call, and the result will be stored in `_4`. This is all the information we need to encode this call, again reusing some of the existing machinery for regular function calls, such as the logic for encoding which permissions need to be exhaled/inhaled during the call.

For the specification functions, we have constructed a dedicated encoder `SpecFunctionEncoder`, which encodes closure contracts and creates the relevant Viper functions (`pre$T()` and `post$T()` so far). Encoding specification entailments happens as part of the existing `SpecEncoder` (which is the module that also encodes quantifiers, implications, *etc.*) by expressing the entailment as quantifiers and calls to the respective specification functions, as explained in Section 5.1.

Higher-order function calls are implemented using the specification rewriting approach (Section 5.1.3), i.e. the higher-order function's specification is rewritten at the call-site to refer to the concrete closure type's set of specification functions.

This concludes our discussion of the Viper encoding techniques and how our implementation automates them. The following chapter will look at examples of user-level specifications for various closures and higher-order functions; in particular, Section 6.2 will demonstrate and discuss the capabilities of our implementation.

6. Evaluation

6.1. Example Specifications

In this section, we will look at how examples from each of the four categories described in Chapter 3 can be equipped with specifications, using the tools presented in Chapter 4.

6.1.1. Higher-Order Functions over Collections

Let us begin with a simple example: The `filter()` method of the `Iterator` trait. For simplicity, though, instead of working with lazy iterators, we will assume that `filter` works on some collection `self.vals` and returns the filtered collection, and for the moment, we will assume that the closure does not modify the elements (which is guaranteed by the postcondition in the specification entailment below). What we want to express is, in the precondition, that we must be able to call the argument closure `f` with every element in the collection, and in the postcondition, that `f` was called on every element, and that its result determined whether the particular element ended up in the resulting collection:

```
// requires: f |= |x| { requires: outer(self.vals).contains(x),
//                      ensures: x == old(x) }
// ensures: forall el: T :: {result.contains(el)}
//     old(self.vals).contains(el) ==>
//     :f (el) / {} ~~> { result == outer(result).contains(el) }
fn filter (self, f: impl FnMut (&T) -> bool) ...
```

With this choice of triggers in the postcondition, we can verify call-site examples such as this one:

```
let nums: Vec<_> = (0 .. 10).collect();
let filtered: Vec<_> =
    nums.into_iter().filter(
        // ensures: result <==> (i % 2 == 0)
        |i| i % 2 == 0).collect();
// assert forall i: i32 :: filtered.contains(i) ==> i % 2 == 0
```

Next, consider the classic `map()` function, operating (for our purposes) on a vector `self.vals` of elements of type `T` and returning a vector of elements of type `U`. We will now lift the restriction from the previous example, i.e. if `T` is a mutable reference, we allow the closure to modify it. It is therefore not sufficient to describe the resulting collection, we also have to describe how the original collection changed, which is illustrated by this example:

```
let mut a = 0;
let mut b = 1;
let mut nums = vec! [&mut a, &mut b];
```

```
let r = nums.iter_mut ()
            .map (|x| { **x += 1; 42 })
            .collect::<Vec<_>> ();
assert_eq! (r, vec! [42, 42]);
assert_eq! ((a, b), (1, 2));
```

Here is a possible specification. Again, we need to require that we can call the closure for every element of `self.vals`. In the postcondition, we can ensure, first, that the result and `self.vals` will have the same length as `old(self.vals)`. This allows us to give a more precise postcondition, because we can now reason about indices (instead of just, say, set membership) and thus about the order of elements in the result. We want to express that `f` has been called with every element from `old(self.vals)`, that its result is at the corresponding index in `result`, and that the poststate value of the element in `self.vals` is what `f` left it in:

```
// requires: f |= |el| { requires: outer(self.vals).contains(el) }
// ensures: old(self.vals).len() == self.vals.len()
// ensures: old(self.vals).len() == result.len()
// ensures: forall idx: usize :: {result[idx]} {self.vals[idx]}
//     0 <= idx && idx < result.len() ==>
//         :f (:x) / { x == outer(old(self.vals[idx])) }
//             ~~> { outer(self.vals[idx]) == x
//                   && outer(result[idx]) == result }
fn map<U, F: FnMut (T) -> U> (&mut self, f: F) -> ...
```

Describing modifications to `f`'s captured state is tricky, because `map()` may call `f` an unbounded amount of times, and so we cannot reason about specific instance values. We can, however, give an invariant as a ghost argument function, which relates the captured state of `f` with the multiset of elements seen so far during `map`'s execution. We can then guarantee `inv(f, old(self.vals))` at the end, but for this to work, we must require that `f` maintains the invariant, and that it is established at the beginning:

```
// ghost_arg: inv: (F, MultiSet[T]) -> bool
// requires: inv(f, {})
// requires: forall s: MultiSet[T] :: {s subset self.vals}
//     s subset self.vals ==>
//         f |= |el| { requires: inv(self, outer(s))
//                               && outer(self.vals).contains(el),
//                     ensures: inv(self, outer(s) union {old(el)}) }
// ensures: inv(f, old(self.vals))
```

This would allow us, for instance, to implement a `fold` in terms of `map`; admittedly, the manual introduction of the closure specification via an assumption in the following example is rather inelegant, but it is necessary: What we would like to express in `f2`'s specification is that it preserves the invariant for all `s` $\subseteq$ `self.vals`. But such a quantification over `s` is not possible in a specification, only with specification entailments, because we can put them under quantifiers. Perhaps future work could improve on this.

```
// ghost_arg: inv: (A, MultiSet[T]) -> bool
// requires: inv(init, {})
// requires forall s: MultiSet[T] :: {s subset self.vals}
```

```
//     s subset self.vals ==>
//         f |= |acc, cur| {
//             requires: inv(acc, outer(s))
//                 && outer(self.vals).contains(cur),
//             ensures: inv(result, outer(s) union {old(cur)})
//         }
// ensures: inv(result, old(self.vals))
fn fold<T, A: Clone> (self, init: A, f: impl Fn(A, &T) -> A) -> A
{
    let mut a = init;

    let mut f2 =
        // view: a: i32 = a
        |el: &T| -> () {
            a = f (a.clone (), el);
        };

    // assume forall s: MultiSet[T] :: {s subset self.vals}
    //     s subset self.vals ==>
    //         f2 |= |el| { requires: inv(self.a, outer(s))
    //                         && outer(self.vals).contains(el),
    //                      ensures: inv(self.a, outer(s) union {old(el)})
    //         }

    // ghost_arg: inv(f2, s) := inv(f2.a, s)
    self.map (f2);

    return a;
}
```

Again, as a simplification, we assume that `map` and `fold` work on `self.vals` instead of lazy iterators. At the call-site, the specification shown above would allow us to verify calls to `fold` such as:

```
let vals = vec! [1, 2, 3, 4];
let acc = vals.into_iter()
              // ghost_arg: inv(x, s) := (x == sum(s))
              .fold(0, // ensures: result == a + c
                       |a, c| a + c);
assert_eq! (acc, 10);
```

Using `fold` to accumulate numbers by addition/multiplication/…is very common in practice. Once support for set comprehensions is made available in Viper (which is an orthogonal issue to closure verification), we will be able to encode examples like these in Viper.

6.1.2. Higher-Order Functions with Fixed Behavior

Consider the `Option<T>::map()` function: It calls its argument closure once with the stored value, if the `Option` stores a value, and never otherwise. Therefore, in case there is no value in the `Option`, we do not have to require anything about the closure's behavior; otherwise, we require that the closure can be called with the stored value and ensure that it has indeed been called:

```
// requires: match self {
//                None => true,
//                Some(val) => f |=! |x| { requires: x == outer(val) }
//            }
// ensures: match old(self) {
//                None => result == None,
//                Some(val) => old(f) (val) / {}
//                    ~~> { outer(result) == Some(result) }
//          }
fn map<U, F: FnOnce (T) -> U> (self, f: F) -> Option<U> { ... }
```

We can also specify function composition. A precise specification for this operation turns out to be surprisingly involved; our strategy here is to expose the captured "sub-closures" `f` and g as views, and to express the result closure's specification in terms of the specifications of `f` and g. Note the use of `ReturnT` to refer to the anonymous existential return type (as in Section 4.4.2).

We do not need a precondition for `compose`—everything is expressed in terms of `result`'s behavior in the postcondition. In the postcondition, we guarantee the initial values of the ghost functions `cf` and `cg`, which are equal to the two closures `f` and g we passed in. The behavior of `result` is then expressed in terms of the behavior of `cf` and `cg`. This allows `compose`'s caller to apply all of her knowledge about `f`'s and g's concrete behavior; she can even reason about precise instance values (as given by the arrows in `result`'s postcondition) and therefore keep track of how the captured state evolves in detail. `result`'s precondition is similar to what we have encountered in Section 4.2.4, i.e. the caller must prove that we can call `cf` (again, we are reasoning about precise instances to give a specification as detailed and powerful as possible) with `result`'s argument, and then `cg` with `cf`'s result.

```
// ghost_result: cf: (&ReturnT) -> &F
// ghost_result: cg: (&ReturnT) -> &G
// ensures: cf(result) == old(f) && cg(result) == old(g)
// ensures:
//     result |= |x| {
//         requires: cf(self) |= |a| {
//             requires: a == outer(x),
//             ensures: cg(outer(self)) |= |b| {
//                 requires: b == outer(result) } },
//         ensures: old(cg(self)) (:b) / {
//                 outer(old(cf(self))) (outer(old(x))) / {}
//                     ~~> { result == outer(b)
//                           && outer(outer(cf(self))) == self } }
//             ~~> { result == outer(result)
//                   && outer(cg(self)) == self } }
fn compose<A, B, C, F: FnMut (A) -> B, G: FnMut (B) -> C>
    (mut f: F, mut g: G) -> impl FnMut (A) -> C
{
    // view: cf: &F = &f
    // view: cg: &G = &g
    move |a: A| g (f (a))
    // ghost_return cf(r) := r.cf
```

```
    // ghost_return cg(r) := r.cg
}
```

6.1.3. `sort_by()`

For `sort_by()`, we need to specify that the comparator closure `cl` is reflexive, antisymmetric, and transitive. Assume, for simplicity, that the comparator closure returns a `bool` (`true` if the comparison of the arguments evaluates to true, `false` otherwise), and that `sort_by()` returns the sorted vector. Note that in the following precondition for `sort_by()`, we not only need to differentiate multiple `results`, but also multiple `old()`s, because each call has its own prestate:

```
// requires: cl |= |a, b| { ensures: old(a == b) ==> result == true }
// requires: cl |= |a, b|, |c, d| { ensures:
//     (old0(a) == old1(d) && old0(b) == old1(c)) && result0 && result1
//         ==> old0(a == b) }
// requires: cl |= |a, b|, |c, d|, |e, f| { ensures:
//     ((old0(b) == old1(c) && old0(a) == old2(e) && old1(d) == old2(f))
//         && result0 && result1) ==> result2 == true }
```

We could also require that `cl` must be "deterministic" (meaning, in this case, that it always returns the same result for the same arguments):

```
// requires: cl |= |a, b|, |c, d| { requires: a == c && b == d,
//                                  ensures: result0 == result1 }
```

This is weaker than requiring `cl` to be pure, because it may still mutate its captured state, as long as the result is deterministic.

Writing a suitable postcondition is harder, but the following should work. The idea is to express the resulting vector in terms of `cl`'s behavior; specifically, if we were to compare `result[i]` with `result[j]` for some $i \leq j$ using the `cl` comparator closure, `cl` should return `true`, because `result` is sorted according to `cl`:

```
// ensures: forall i, j :: {result[i], result[j]}
//     0 <= i <= j < result.len() ==>
//         cl |= |a, b| { requires: a == outer(result[i])
//                                  && b == outer(result[j]),
//                        ensures: result == true }
```

This would allow us to verify call-sites such as:

```
let cl = |a: &i32, b: &i32| *a <= *b;
let mut nums = vec! [3, 2, 1];
assert! (cl (&nums[0], &nums[1]) == false);
nums.sort_by (cl);
assert! (cl (&nums[0], &nums[1]));
```

Additionally, we should ensure that `result` is a permutation of `self.vals`, i.e. contains the same elements. This could be expressed as follows, assuming the availability of a `to_multiset` operator that returns a multiset containing the elements in a given vector:

```
// ensures: self.vals == old(self.vals)
// ensures: to_multiset(self.vals) == to_multiset(result)
```

6.1.4. Boxed Closures

Consider this example from the Rust Book [16, Section 13.1], adapted to use a boxed closure rather than a generic type parameter. The idea of this (artificial) example is that `Cacher` stores a closure and an option, and only calls the closure if there is not already a value stored in the option (in which case it will put the closure's result into the option):

```
struct Cacher
{
    calculation: Box<dyn Fn(u32) -> u32>,
    value: Option<u32>,
}

impl Cacher
{
    fn new(calculation: Box<dyn Fn(u32) -> u32) -> Cacher {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

Note that even though `Cacher`'s fields are not declared `pub` (public), code in the same module can still access and modify them, so for our purposes, the visibility of those fields does not really make a difference. In particular, the `value` field can be set back to `None` at any time, the `calculation` field can be reassigned, *etc.*

We will present two different specification strategies for this example. Either one might be the more appropriate solution, depending on the concrete use case. The first one leaves `Cacher` unchanged and equips the two functions with the following specifications:

```
// ensures: result.calculation == old(calculation)
// ensures: result.value == None
fn new (calculation: Box<dyn Fn(u32) -> u32) -> Cacher { ... }

// requires: match self.value {
//     None => self.calculation |= |i| { requires: i == outer(arg) },
//     Some(_) => true
```

```
// }
// ensures: match old(self.value) {
//     Some(v) => (self.value == Some(v) && result == v
//                 && self.calculation == old(self.calculation)),
//     None => (old(self.calculation) (arg) / {}
//                 ~~> { result == outer(result)
//                       && outer(self.value) == Some(result)
//                       && outer(*self.calculation) == self })
// }
fn value(&mut self, arg: u32) -> u32 { ... }
```

Here, it is the caller's responsibility to keep track of what's stored in `Cacher::calculation`; in particular, `value()` requires a specification entailment in its precondition.

The second strategy adds an invariant to `Cacher`:

```
struct Cacher {
    // invariant: calculation |= |i| {
    //     requires: i >= 10, ensures: result <= 100
    // }
    // invariant: match value {
    //     Some(i) => i <= 100,
    //     None => true
    // }
    ...
}

// requires: calculation |= |i| {
//     requires: i >= 10, ensures: result <= 100 }
// ensures: result.value == None
fn new (calculation: Box<dyn Fn(u32) -> u32) -> Cacher { ... }

// requires: arg >= 10
// ensures: result <= 100
fn value(&mut self, arg: u32) -> u32 { ... }
```

This is a much more concise specification. It is also more modular in the sense that the caller no longer has to keep track of which closure is stored in the `Cacher::calculation` field—instead, the invariant guarantees that any closure that might be stored in that field will always have a certain behavior.

On the downside, this specification is less flexible (because every closure we might want to store in the `calculation` field has to obey the given specification) and less precise: We might know a stronger specification for the closure we pass into `new()`, but this information will be lost by the time we call `value()`. Also, `value()` does not specify how its result is calculated, i.e. whether it actually implements a caching behavior.

Again, it will be up to the programmer writing the specification to decide which of the two strategies better fits her particular purposes. A trade-off has to be made between flexibility and precision on the one hand and modularity, information hiding, and simplicity of the specification on the other hand. Our methodology supports both strategies.

6.2. Our Implementation

Our implementation allows closures to be defined, equipped with specifications, and called. For instance, the following program

```
extern crate prusti_contracts;
use prusti_contracts::*;

fn test() {
    let f = closure! (
        requires (a > b),
        ensures (result > b),
        |a: i32, b: i32| -> i32 { a }
    );

    let a = 5;
    let b = 3;
    let c = f (a, b);
    assert! (c > b);
}

fn main() {}
```

verifies with

```
Verification of 3 items...
Successful verification of 3 items
```

Changing the closure call to, say, `f(b, a)` correctly results in a verification error:

```
Verification of 3 items...
error: [Prusti: verification error] precondition might not hold.
  --> /tmp/cl.rs:13:13
   |
13 |     let c = f (b, a);
   |             ^^^^^^^^
   |
note: the failing assertion is here
  --> /tmp/cl.rs:6:19
   |
6  |         requires (a > b),
   |                   ^^^^^

Verification failed
error: aborting due to previous error
```

Our implementation also handles specification entailments. As an example, we are able to correctly verify this program, containing a closure specification as well as a specification entailment that strengthens the closure's precondition and weakens the postcondition, assuming the strengthened precondition:

```
extern crate prusti_contracts;
use prusti_contracts::*;

fn test () {
    let f = closure! (
        requires (i >= 0),
        ensures (result == i + 1),
        |i: i32| -> i32 { i + 1 }
    );

    test2 (f);
}

#[requires(f |= |i: i32| [ requires(i >= 5), ensures(result >= 6) ])]
fn test2<F: Fn (i32) -> i32> (f: F) {}

fn main () {}
```

Changing the precondition of `test2()` to, say,

```
#[requires(f |= |i: i32| [ requires(i >= -1), ensures(result >= 0) ])]
```

which strictly weakens the precondition of `f` when calling `test2`, correctly results in a verification error:

```
Verification of 4 items...
error: [Prusti: verification error] precondition might not hold.
  --> /tmp/specent.rs:11:5
   |
11 |     test2 (f);
   |     ^^^^^^^^^
   |
note: the failing assertion is here
  --> /tmp/specent.rs:14:12
   |
14 | #[requires(f |= |i: i32| [ requires(i >= -1), ensures(result >= 0) ])]
   |            ^                        ^^^^^^^           ^^^^^^^^^^^

Verification failed
error: aborting due to previous error
```

Note how the error report points to the precondition of `test2`, which is correct, since the specification entailment is part of that function's precondition.

Calling `f` from inside `test2` does not work yet, because the permission transfer to and from `f` before/after the call is not yet handled properly. In particular, because `f` is of generic type, we cannot know its precise signature (which includes the implicit `self` argument). The rest of the signature could be read off the type bound (`F: Fn (i32) -> i32`, i.e. the signature modulo the `self` argument is `|i32| -> i32`), but even this is non-trivial, because giving contradictory type bounds is valid Rust code:

```
fn foo<T> (f: T)
    where T: (Fn (i32) -> i32) + (Fn (String) -> Option<i32>)
{}
```

Here, `T` must be a subtype of *both* the specified traits (conjoined with the `+` operator), meaning it must have two signatures (`|i32| -> i32` and `|String| -> Option<i32>`) simultaneously. Such a type does not exist, but generic types need not be proven to be inhabited in Rust; it is the *caller's* responsibility to instantiate the generic parameters, hence `foo` above would compile without errors.

Views for the captured state (and, consequently, invariants), ghost arguments, and the arrow notation have not been implemented yet due to time constraints. Some of them should be a rather straightforward addition, though—for instance, the arrow notation should be easy to implement because the specification functions are already implemented and working. Furthermore, we have already explained possible implementation strategies for all of these features in Chapter 5; thus, the remaining work on the implementation should mostly consist of engineering (rather than research) problems.

7. Conclusion

Modern automatic program verifiers are often based on first-order logics, which makes the integration of higher-order functional concepts, such as closures as first-class function-type objects, not straightforward. Nevertheless, we have demonstrated that extending a standard first-order specification language with a few relatively simple operators and notions suffices to describe and prove a range of practical examples of closure occurrences. Still, some issues remain, which is not surprising considering the wide variety of applications for closures and higher-order functions, as well as the intrinsic difficulty of the problem that lies in their higher-order nature. For this reason, the following section suggests some opportunities for future research in this domain.

Future Work

This thesis has aimed to capture and highlight many ideas related to closure verification, but not all of them have been developed fully. For instance, the ghost state extension from Section 4.1.4 has only relatively briefly been discussed; further thought and research might be necessary to understand their utility, limitations, and the feasibility of their implementation better. In particular, the *integration* of this idea with the rest of our approach should be studied more deeply; for instance, one could attempt to combine this ghost state with the ghost arguments and results from Section 4.4.

Moreover, not all of the features presented here have actually been implemented in Prusti so far. Future work could therefore focus on extending the implementation, at least to the extent desired by the maintainers.

Although this thesis has focused on Rust and occasionally relied on guarantees made by its type system—for instance, preservation of the history invariants of Section 4.1.3 only works[1] because Rust does not allow mutable references to the captured state from outside the closure while the closure instance is live—, the basic ideas should nonetheless be applicable to and valuable for verification efforts in other languages. Another avenue for future research thus lies in the adaptation of the tools presented here for different programming languages and verification toolstacks.

And finally, as noted above, this thesis does of course not comprehensively solve all issues related to closure and higher-order function verification. For instance, side-effectful closures occurring as arguments to the higher-order functions of Section 3.1 are not yet sufficiently covered by our approach in case their behavior depends on the *order* of the calls. Therefore, future work could investigate such limitations and weaknesses of our approach and try to overcome them, while simultaneously attempting to simplify the user-level specification language to allow for even more concise, expressive, and readable specifications.

[1] At least without additional reasoning and restrictions.

A. Proofs about Weakened Relations

Here is a simple Coq script that demonstrates that neither pseudo-reflexivity (see Section 5.1) nor transitivity are necessarily preserved when weakening pseudo-reflexive and transitive relations. Theorem `weaken_refl` shows that pseudo-reflexive and transitive relations can be weakened so as not to be pseudo-reflexive anymore, and `weaken_trans` shows that transitivity is not necessarily preserved either.

```
From Coq Require Import omega.Omega.

Definition Rel (T: Type) := T -> T -> Prop.
Definition pseudo_refl {T: Type} (R: Rel T) := forall x: T,
    (exists y: T, R y x) -> R x x.
Definition trans {T: Type} (R: Rel T) := forall x y z: T,
    (R x y) -> (R y z) -> (R x z).


Definition A (a: nat) (b: nat) := (a > 0) /\ (b > 0) /\ (a <= b).
Definition B (a: nat) (b: nat) :=
    ((a > 0) /\ (b > 0) /\ (a <= b)) \/ (a = 1 /\ b = 0).

Lemma Arefl: pseudo_refl A.
Proof.
  unfold pseudo_refl, A. intros x [y H].
  omega.
Qed.

Lemma Bnrefl: ~ (pseudo_refl B).
Proof.
  unfold pseudo_refl, B. intro.
  pose proof (H 0).
  destruct H0.
  - exists 1. right. omega.
  - omega.
  - omega.
Qed.

Lemma Atrans: trans A.
Proof.
  unfold trans, A. intros. omega.
Qed.

Lemma Btrans: trans B.
Proof.
```

```
  unfold trans, B. intros. omega.
Qed.

Lemma B_weakens_A: forall (a b: nat), (A a b) -> (B a b).
Proof.
  unfold A, B. intros. omega.
Qed.

Theorem weaken_refl:
  ~ (forall (T: Type) (R1: Rel T) (R2: Rel T),
        (pseudo_refl R1) -> (trans R1)
        -> (forall x y: T, (R1 x y) -> (R2 x y))
        -> (pseudo_refl R2)).
Proof.
  intro. pose proof (H nat A B Arefl Atrans B_weakens_A).
  pose proof Bnrefl. auto.
Qed.


Definition R (a: nat) (b: nat) := a <= b.
Definition S (a: nat) (b: nat) := (a <= b) \/ (a = 6 /\ b = 4).

Lemma Rrefl: pseudo_refl R.
Proof.
  unfold pseudo_refl, R. intros. apply Nat.le_refl.
Qed.

Lemma Rtrans: trans R.
Proof.
  unfold trans, R. intros. eapply Nat.le_trans; eassumption.
Qed.

Lemma S_weakens_R: forall (a b: nat), (R a b) -> (S a b).
Proof.
  unfold R, S. intros. left. assumption.
Qed.

Lemma Sintrans: ~ (trans S).
Proof.
  unfold trans. intro.
  pose 6 as n. pose 4 as m. pose 5 as o.
  assert (S n m).
  { unfold S. omega. }
  assert (S m o).
  { unfold S. omega. }
  pose proof (H n m o H0 H1).
  unfold S in H2. subst n m o.
  destruct H2; omega.
Qed.
```

```
Theorem weaken_trans:
  ~ (forall (T: Type) (R1: Rel T) (R2: Rel T),
        (pseudo_refl R1) -> (trans R1)
        -> (forall x y: T, (R1 x y) -> (R2 x y))
        -> (trans R2)).
Proof.
  intro. pose proof (H nat R S Rrefl Rtrans S_weakens_R).
  pose proof Sintrans. auto.
Qed.
```

Bibliography

[1] Vytautas Astrauskas et al. "Leveraging Rust types for modular specification and verification". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), pp. 1–30. DOI: 10.1145/3360573.

[2] Ernie Cohen et al. "Local Verification of Global Invariants in Concurrent Programs". In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Red. by David Hutchison et al. Vol. 6174. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 480–494. DOI: 10.1007/978-3-642-14295-6_42.

[3] Ernie Cohen et al. *Verifying C Programs: A VCC Tutorial*. Working draft, version 0.2. July 10, 2015. URL: https://bit.ly/32BkCWN (visited on 09/16/2020).

[4] Robert L. Constable and Anne Trostle. *Logical Investigations, with the Nuprl Proof Assistant, Chapter 5: First-Order Logic: All and Exists*. July 2014. URL: http://www.nuprl.org/MathLibrary/LogicalInvestigations/all_exists.html (visited on 09/16/2020).

[5] Ádám Darvas and K. Rustan M. Leino. "Practical Reasoning About Invocations and Implementations of Pure Methods". In: *Fundamental Approaches to Software Engineering*. Ed. by Matthew B. Dwyer and Antónia Lopes. Vol. 4422. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 336–351. DOI: 10.1007/978-3-540-71289-3_26.

[6] K.K. Dhara and G.T. Leavens. "Forcing behavioral subtyping through specification inheritance". In: *Proceedings of IEEE 18th International Conference on Software Engineering*. Berlin, Germany: IEEE Comput. Soc. Press, 1996, pp. 258–267. DOI: 10.1109/ICSE.1996.493421.

[7] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. "The spirit of ghost code". In: *Formal Methods in System Design* 48.3 (June 2016), pp. 152–174. DOI: 10.1007/s10703-016-0243-x.

[8] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Red. by David Hutchison et al. Vol. 7792. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

[9] Robert Bruce Findler and Matthias Felleisen. "Contracts for higher-order functions". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming - ICFP '02*. Pittsburgh, PA, USA: ACM Press, 2002, pp. 48–59. DOI: 10.1145/581478.581484.

[10] S. M. German, E. M. Clarke, and J. Y. Halpern. "Reasoning about procedures as parameters". In: *Logics of Programs*. Ed. by Edmund Clarke and Dexter Kozen. Red. by G. Goos et al. Vol. 164. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 206–220. DOI: 10.1007/3-540-12896-4_365.

[11] *Guide to Rustc Development*. URL: https://rustc-dev-guide.rust-lang.org/ (visited on 08/16/2020).

[12] Thomas Hader. *Proposal for supporting closures in Prusti*. Aug. 2019. URL: https://bit.ly/2Ry7ZGj (visited on 04/21/2020).

[13] K. Honda, N. Yoshida, and M. Berger. "An Observationally Complete Program Logic for Imperative Higher-Order Frame Rules". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. Chicago, IL, USA: IEEE, 2005, pp. 260–279. DOI: 10.1109/LICS.2005.5.

[14] Johannes Kanig and Jean-Christophe Filliâtre. "Who: a verifier for effectful higher-order programs". In: *Proceedings of the 2009 ACM SIGPLAN workshop on ML - ML '09*. Edinburgh, Scotland: ACM Press, 2009, p. 39. DOI: 10.1145/1596627.1596634.

[15] Ioannis T. Kassios and Peter Müller. *Specification and verification of closures*. ETH Zurich, 2010. DOI: 10.3929/ETHZ-A-006843251.

[16] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ (visited on 09/18/2020).

[17] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. Advisor: Aldrich, Jonathan. 2012. URL: http://reports-archive.adm.cs.cmu.edu/anon/anon/home/ftp/2012/CMU-CS-12-127.pdf (visited on 05/07/2020).

[18] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4 (Jan. 1, 1964), pp. 308–320. DOI: 10.1093/comjnl/6.4.308.

[19] K. Rustan M. Leino. *This is Boogie 2*. Manuscript KRML 178, working draft 24 June 2008. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf (visited on 07/04/2020).

[20] K. Rustan M. Leino and Rosemary Monahan. *Automatic verification of textbook programs that use comprehensions*. June 30, 2007. URL: http://mural.maynoothuniversity.ie/3935/ (visited on 04/29/2020).

[21] K. Rustan M. Leino and Peter Müller. "Verification of Equivalent-Results Methods". In: *Programming Languages and Systems*. Ed. by Sophia Drossopoulou. Vol. 4960. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 307–321. DOI: 10.1007/978-3-540-78739-6_24.

[22] K. Rustan M. Leino and Wolfram Schulte. "Using History Invariants to Verify Observers". In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Red. by David Hutchison et al. Vol. 4421. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 80–94. DOI: 10.1007/978-3-540-71316-6_7.

[23] Barbara H. Liskov and Jeannette M. Wing. "A behavioral notion of subtyping". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (Nov. 1994), pp. 1811–1841. DOI: 10.1145/197320.197383.

[24] Joel Moses. *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should be Called the Environment Problem*. AI Memo 199. June 1, 1970. URL: http://hdl.handle.net/1721.1/5854 (visited on 07/05/2020).

[25] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

[26] Martin Nordio et al. "Reasoning about Function Objects". In: *Objects, Models, Components, Patterns*. Ed. by Jan Vitek. Red. by David Hutchison et al. Vol. 6141. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 79–96. DOI: 10.1007/978-3-642-13953-6_5.

[27] Yann Régis-Gianas and François Pottier. "A Hoare Logic for Call-by-Value Functional Programs". In: *Mathematics of Program Construction*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 305–335. DOI: 10.1007/978-3-540-70594-9_17.

[28] John C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. Copenhagen, Denmark: IEEE Comput. Soc, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[29] *Rust by Example*. URL: https://doc.rust-lang.org/stable/rust-by-example/ (visited on 07/04/2020).

[30] Malte Schwerhoff and Alexander J. Summers. "Lightweight Support for Magic Wands in an Automatic Verifier". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 614–638. DOI: 10.4230/LIPIcs.ECOOP.2015.614.

[31] Alexander J. Summers and Peter Müller. "Automating deductive verification for weak-memory programs (extended version)". In: *International Journal on Software Tools for Technology Transfer* (Mar. 6, 2020). DOI: 10.1007/s10009-020-00559-y.

[32] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. "Verifying Generics and Delegates". In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by Theo D'Hondt. Red. by David Hutchison et al. Vol. 6183. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 175–199. DOI: 10.1007/978-3-642-14107-2_9.

[33] Nikhil Swamy et al. "Verifying higher-order programs with the dijkstra monad". In: *ACM SIGPLAN Notices* 48.6 (June 23, 2013), pp. 387–398. DOI: 10.1145/2499370.2491978.

[34] *The Rust Standard Library*. URL: https://doc.rust-lang.org/std/index.html (visited on 07/04/2020).

[35] D. A. Turner. "Some History of Functional Programming Languages". In: *Trends in Functional Programming*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Red. by David Hutchison et al. Vol. 7829. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–20. DOI: 10.1007/978-3-642-40447-4_1.

[36] *Viper Tutorial*. URL: http://viper.ethz.ch/tutorial/ (visited on 07/04/2020).

[37] Benjamin Weber. *Automating Modular Reasoning About Higher-Order Functions*. Master's Thesis. Nov. 12, 2017. URL: https://bit.ly/2zkCbhG (visited on 04/23/2020).