Modular specification and verification of closures in Rust

Modular Specification and Verification of Closures in Rust

FABIAN WOLFF, ETH Zürich, Switzerland
AUREL BÍLÝ, ETH Zürich, Switzerland
CHRISTOPH MATHEJA, ETH Zürich, Switzerland
PETER MÜLLER, ETH Zürich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada

Closures are a language feature supported by many mainstream languages, combining the ability to package up references to code blocks with the possibility of capturing state from the environment of the closure's declaration. Closures are powerful, but complicate understanding and formal reasoning, especially when closure invocations may mutate objects reachable from the captured state or from closure arguments.

This paper presents a novel technique for the modular specification and verification of closure-manipulating code in Rust. Our technique combines Rust's type system guarantees and novel specification features to enable formal verification of rich functional properties. It encodes higher-order concerns into a first-order logic, which enables automation via SMT solvers. Our technique is implemented as an extension of the deductive verifier Prusti, with which we have successfully verified many common idioms of closure usage.

CCS Concepts: • **General and reference** → **Verification**; • **Theory of computation** → *Programming logic*; *Functional constructs*; **Program specifications**; **Program verification**; Invariants; • **Software and its engineering** → **Software verification**; **Formal software verification**.

Additional Key Words and Phrases: Rust, closures, higher-order functions, software verification

ACM Reference Format:
Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular Specification and Verification of Closures in Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 145 (October 2021), 29 pages. https://doi.org/10.1145/3485522

1 INTRODUCTION

Although dating back to at least 1964 [Landin 1964], the programming language community has seen a renewed interest in closures as a language feature this millennium, with their addition to many imperative and object-oriented mainstream languages, including C++ (in C++11), Java (v. 8), and C# (v. 3.0) [Mazinanian et al. 2017]. Closures allow for the encapsulation of code fragments as functions that – in contrast to an object's methods – can be passed around as first-class values without revealing the existence of associated state. For example, a `filter` function in a data structure API might take a closure as argument, allowing callers to instantiate the filtering criterion; a closure passed to `filter` might store in its associated state how many times the criterion was invoked. We refer to functions (such as `filter`) taking closures as arguments as *higher-order functions*.

Authors' addresses: Fabian Wolff, ETH Zürich, Department of Computer Science, Zürich, Switzerland, fabian.wolff@alumni.ethz.ch; Aurel Bílý, ETH Zürich, Department of Computer Science, Zürich, Switzerland, aurel.bily@inf.ethz.ch; Christoph Matheja, ETH Zürich, Department of Computer Science, Zürich, Switzerland, cmatheja@inf.ethz.ch; Peter Müller, ETH Zürich, Department of Computer Science, Zürich, Switzerland, peter.mueller@inf.ethz.ch; Alexander J. Summers, University of British Columbia, Department of Computer Science, Vancouver, Canada, alex.summers@ubc.ca.

Closures in purely functional languages offer a convenient mechanism for writing functions without explicitly listing all their arguments; a closure may capture variables from its environment, but since those variables are immutable, calling such a closure has no side effects on its captured state: subsequent invocations of the same closure instance always yield the same behaviour.

By contrast, closures in imperative and object-oriented languages combine *side-effectful* implementations, *aliasing*, and *mutable captured state;* a closure may capture references to pre-existing data when declared, and calling such a closure can have side effects on both this captured state and any arguments passed in the call. This delicate combination makes it difficult to precisely understand code manipulating closures, and presents challenges for modular reasoning about such code, whether informally in, e.g. a code review, or formally via program verification.

In the presence of aliasing, informal reasoning about imperative code is error-prone, and mistakes lead naturally to memory errors, runtime exceptions, data races, or simply wrong results being silently produced. Formal reasoning techniques addressing these challenges in general, such as advanced program logics [Harel et al. 2002; Kassios 2006; O'Hearn et al. 2001; Smans et al. 2012], are powerful but complex, and their application is typically restricted to verification experts. Moreover, existing verification techniques for reasoning about closures [Kanig and Filliâtre 2009; Krishnaswami 2012; Svendsen et al. 2010; Yoshida et al. 2007] typically require explicit proofs, often in a higher-order logic, or support no or limited usage of mutable (captured) state. Working in a higher-order logic makes the task of writing and understanding specifications substantially more sophisticated for a user, while drastically limiting the potential for their automated verification.

In this paper, we present a novel technique for the modular specification and verification of closures in Rust [Matsakis and Klock 2014]. Rust's ownership type system controls aliasing and side effects, and rules out a wide variety of common errors such as dangling pointers and data races by design. Among our key insights is that we can derive strong framing guarantees from Rust's type system: for every Rust closure, we know that *no captured state* can *ever* be mutated except by the closure itself – a property far stronger than aliasing or mutability guarantees for closures in other (imperative) languages. Extracting framing information for closures is therefore an essential ingredient of our verification technique, but it is not the only one. Equally importantly, we introduce novel specification features to enable succinct specification and verification of multi-state properties of closures without breaking modularity or leaving the realm of first-order logic. It is the combination of leveraging Rust's framing guarantees, concise specification constructs geared towards modular verification, and staying encodable in a first-order setting that truly pushes automated verification.

Despite our primary focus on Rust, our verification technique is in principle compatible with other languages with ownership type systems, and even those without if combined with a suitable underlying program logic for heap reasoning.

Contributions. The main contributions of our work are:

(1) An informal analysis of existing Rust code, surveying common use cases for closures in practice and identifying the relevant verification challenges arising therefrom (→ Section 2, with some experimental justifications in Section 5.3).
(2) A novel specification technique enabling the expression of rich properties related to closures while providing strong support for modular and automated verification, even in the presence of side effects on closure arguments and captured state (→ Section 3).
(3) An encoding for our technique, mapping the higher-order verification problem down to constraints which can be expressed in a first-order setting, suitable for automation using an SMT-based verification toolchain (→ Section 4).

```rust
fn foo(v: Vec<i32>) {
    let mut count: i32 = 0;
    let f = |x: &i32| *x * 2;
    let g = |x: &i32| { count = count + 1; *x + 3 };
    let u: Vec<i32> = v.iter().map(f).collect();
    let w: Vec<i32> = v.iter().map(g).collect(); ... }
```

Fig. 1. Simple Rust example illustrating closure syntax

(4) An implementation of our technique as an extension of a state-of-the-art Rust verifier [Astrauskas et al. 2019], and an evaluation which demonstrates that our technique can handle common use cases of closures in practical Rust code (⟶ Section 5).

2 RUST CLOSURES

This section presents three central aspects of background for our work: (1) our overall design goals, (2) common use cases of closures and the challenges they pose for closure and higher-order function verification, and (3) how Rust's language design and type system present new opportunities for addressing these challenges with a more lightweight and more automatable specification technique.

2.1 Overarching Design Goals

Before diving into Rust closures and their uses in practice, we will present the two main design goals that underlie the challenges of closure and higher-order function verification.

2.1.1 Modularity. Modular specification and verification techniques allow one to reason about program components independently, without requiring access to the implementation of other components, such as client code. Modularity is essential for verification to scale and for providing strong guarantees for individual components such as libraries. However, modularity is difficult to achieve in the context of closures and higher-order functions, especially in the presence of mutable state. Consider first the example snippet of Fig. 1 introducing basic Rust syntax. In Rust, **`let`** is used to declare variables; types can be inferred automatically but **`mut`** is necessary on those variables which may later be mutated.[1] A closure is declared using `|args| {body}` syntax (braces around the body can be omitted for simple cases), e.g. `f` stores a closure taking a reference to an integer and returning twice its referenced value. Next, `g` stores a closure taking a similar parameter, but also performing side effects on the captured `count` variable each time the closure is called. The code then passes each closure to a `map` higher-order function (the other calls are concerned with Rust iterator syntax, and are not our focus here), applying the closure to each element to produce the new vector's contents. Upon completion, each element of vector `u` equals twice the value at the same position in vector `v`, and each element of `w` is equal to the corresponding element in `v` plus three; `v` itself remains unchanged, and `count` stores the size of vector `v`.

Although very simple, this example already serves to illustrate the challenges of modular reasoning about such code. We aim to enable independent specifications on closures such as `f` and `g` and higher-order functions such as `map` which can be *composed* when reasoning about calls such as the two above to formally summarise the results and resulting side effects. Our methodology will

[1]To improve readability, we added type annotations to variables that are not strictly necessary in Rust. Additionally, our implementation (⟶ Section 5) requires type annotations on all closure arguments even though the compiler can infer them automatically; this limitation is present in the tool we build upon, which does not yet have access to the types inferred by the compiler when processing specification annotations.

neither special-case particular higher-order functions, nor rely on more than their specifications and those of closures passed to them to reason about each call: we present a *modular* verification methodology that verifies higher-order functions once and for all, independently of client code.

2.1.2 Automatability. In order for our methodology to be compatible with standard approaches to automated program verification, we must be able to encode its proof obligations into a suitable (first-order) logic, e.g. to enable SMT-based tooling. The fact that we are dealing with higher-order concepts makes this a significant challenge (which, to our knowledge, has not been tackled in general in prior work). In particular, we will need an encoding of our methodology which does not rely on direct quantification over closures, their specifications, their calls, or states in which they are invoked. These notions are highly relevant for specification, but perhaps surprisingly we will show how to encode these down to standard verification conditions suitable for automation.

2.2 Basic Usage of Closures

In this subsection, we review some basic properties of closures and distil the key problems that we address in the remainder of this paper.

2.2.1 What Effects Can Rust Closures Have? As we have seen, Rust closures can have arguments, a return value, and captured state. A closure call's observable effects can manifest in all three of these: during a call, a closure may mutate arguments passed by mutable reference, a closure may mutate its captured state, and it may produce a result, as in the following example:

```
let mut i = 1;
let mut j = 2; // j is captured by (mutable) reference below:
let mut cl = |x: &mut i32| -> i32 { j += *x; *x += 1; j + *x };

let r = cl(&mut i);
assert_eq!(i, 2); // the closure's side effect on its argument
assert_eq!(j, 3); // the closure's side effect on its captured state
assert_eq!(r, 5); // the closure's result
```

To prove the above assertions, we need to reason about the closure's result in terms of its argument and captured state, as well as its side effects on each of these. The definition of `cl` captures `j` (by mutable reference, the default for closure-captured state which is modified in the closure body). In this simple case, one could imagine simply inlining the closure call, but in general (e.g. when a closure is passed as a function argument), we need to be able to modularly prove correctness of the closure body and capture sufficient information about its behaviour via a suitable specification. The first step towards achieving this goal is tackling the following first challenge:

Key Problem K1: How to specify guaranteed closure behaviour for all future call sites, potentially conditioned on values of the closure's captured state.

2.2.2 How May the Captured State Evolve? Similar to other aggregate types, closures typically do not modify their captured state arbitrarily, but rather (implicitly) establish and maintain invariants on the values this state can take. For example, consider the following closure `inc`:

```
let mut x = 0;
let mut inc = || { x += 1; x };
```

Clearly, `inc` will always return positive values, however many times it is called (assuming no overflow occurs). Formally proving this property, however, involves establishing the *invariant*

$x \geq 0$ on inc's captured state. More generally, exactly *which* properties are guaranteed for all future calls of a particular closure may depend both on knowledge of the closure's initially captured state and on *previous* calls to the same closure instance, as illustrated by the following example:

```
use std::collections::HashSet;
let mut set = HashSet::new();        // mutable standard library set
set.insert(42);                      // inserts 42 into the set
let mut cl = |i| { set.insert(i); }; // cl captures set (mutably)
cl(7);                               // (indirectly) inserts 7 into the set
foo(cl);                             // pass cl to some higher-order function
// (cl is no longer live here, releasing alias to set from its captured state)
assert!(set.contains(&42) && set.contains(&7));
```

Without knowing foo's precise behaviour (in particular, whether and how often it calls its argument closure), we would like to assert that both elements remain in the set. Conceptually, this is guaranteed without needing to know anything about foo's behaviour: the implementation of cl only ever adds to the set, while Rust's type system prevents there being any other usable alias to the same set while the closure is live. This illustrates the next key challenge:

Key Problem K2: Expressing the possible evolution of a closure's captured state across unknown, potentially unbounded numbers of calls to the closure.

2.3 Common Closure Use Cases

Closures have a wide variety of idiomatic use cases; we give an overview of several of the more common motivations for closure usage in Rust (and similar languages) here. These impose minimum requirements on the expressiveness of our specification and verification technique.

2.3.1 Pointwise Computations. The following example is, albeit simple, typical of real-world Rust code: take a collection and compute from each element a corresponding output element, each computed by an (independent) call to the same closure. The specific computation performed per element can be customised by varying the types and definition of the closure, e.g. mapping integers to their absolute values, parsing strings as floating-point numbers, etc.

```
let nums = vec![1, 2, 3];
let r = nums.iter()            // creates an iterator over the vector
            .map(|i| (*i) * 2)
            .collect::<Vec<_>>(); // collects the iterator's elements into a vector
assert_eq!(r, vec![2, 4, 6]);
```

Apart from Iterator::map(), several other standard library functions, such as Iterator::for_each() and Option::map(), also implement similar pointwise behaviours. To verify such examples, the specification of, say, map() must express its behaviour in terms of the closure it receives as an argument. Indeed, this is true for any higher-order function: the specification of a higher-order function may need to express *requirements* on the closure's behaviour, ahead of any concrete calls (such as "the closure must always return positive values"), and may (in retrospect) also summarise the actual calls made to its closure argument(s), and how these relate to the results and side effects of the higher-order function called (e.g. "the closure was called with argument X, and the result of this call was stored in collection Y"). This duality is captured by the next two key problems:

Key Problem K3: Specifying higher-order programming idioms, such as those implemented by commonly used higher-order functions in the Rust standard library, including suitable requirements on closure values passed to such functions.

Key Problem K4: Specifying the side effects and results of higher-order functions in terms of properties of the closures they are passed.

2.3.2 Closures as Lazy Generators. Closures are often used for generating values from scratch (for instance, in order to initialise data structures, or to provide default values). The value of this pattern lies in its making element creation *lazy:* elements are only constructed (by calling the closure) when actually needed. Examples from the standard library include the `unwrap_or_else()` function of `Result` and `Option` (and, in fact, any of the many standard library functions with names ending in `_or_else()`). Additionally, the `repeat_with()` function illustrates this pattern nicely: it receives a closure argument and constructs an infinite iterator containing the results of closure invocations:

```rust
use std::iter::repeat_with;
let mut count = 0;
let nums = repeat_with(
    || { let r = count; count += 1; r * 2 });

for i in nums.take(10) { // iterates over the first 10 values
    assert!(i % 2 == 0);
}
```

Often, this is used in conjunction with calls to the lazy function `take()` (to obtain a finite iterator) and the non-lazy function `collect()` (to insert the iterator's elements into a new data structure). For this use case, one typically wants to verify statements of the form "all generated values have a certain property"; similarly to pointwise computations above, key problems K3 and K4 arise.

2.3.3 Closures as Accumulators. In addition to the simple use cases from the previous two paragraphs, closures can also be used to control more-complex accumulations, in which each call may depend on previous ones. A classic example is the `fold()` function, which threads an accumulated value through all calls to its argument closure:

```rust
let nums = vec![1, 5, 7, 11, 13];
let all_odd = nums.iter()
                  .fold(true, |a, c| a && (c % 2 == 1));

if all_odd {
    for i in nums {
        assert!(i % 2 == 1);
    }
}
```

`fold()` makes the accumulation explicit, but examples involving `for_each()` and even `map()` (all from the `Iterator` trait) can also fall into this category whenever their argument closure performs accumulations *implicitly*, in its captured state. For all such cases, the following challenge arises:

Key Problem K5: Specifying aggregated values computed via the combination of repeated closure calls.

2.3.4 Closures as Predicates. Previous examples of closure usage did not affect the control flow – the closures we presented were straight-line programs determining, for example, the operations to be applied to each element of a vector. By contrast, we now address cases where the main purpose of a closure is to influence the control flow of a higher-order function. For instance, `sort_by()` takes a closure as an argument that determines the sorting criterion. Similarly, the `take_while()` and `skip_while()` functions of `Iterator` receive an argument closure corresponding, conceptually, to a loop *head*, because the closure determines how many elements should be retained or skipped, respectively. The following example illustrates this use case with the `skip_while()` function, skipping over elements of a vector until the first negative value is reached:

```
let nums = vec![1, 2, 3, -1];
let x = nums.iter().skip_while(|i| **i >= 0).next();
assert_eq!(x, Some(&-1));
```

Reasoning about such code typically relies implicitly on properties such as anti-symmetry of a comparison function, or transitivity of an equivalence property expressed via a closure; we need mechanisms for describing these properties of multiple arbitrary calls to the closure.

Key Problem K6: Specifying the mathematical relations of closures acting as predicates.

2.3.5 Closures for Concurrency. Since closures provide a convenient abstraction over a block of code, they are also used as a standard mechanism for concurrency libraries such as `std:thread`, whose `spawn` function takes a closure as argument to specify the code to be executed. While formally reasoning about such concurrent programs requires many of the solutions we present in this paper for handling the closure-related reasoning, verification of concurrent Rust is beyond the direct scope of our work here.

2.4 Rust Specifics

All of the use cases presented so far equally occur in programming languages other than Rust, so what makes closures in Rust specifically interesting? In this subsection, we argue that Rust simplifies some aspects of reasoning about such closure uses thanks to the specifics of its type system. Solving the key problems described above would be harder still in other languages, due to the possibility of *unrestricted aliasing*. For example, reasoning challenges exist for tracking relevant information about a closure's captured state and how it evolves (see key problems K1–K4). However, *for every Rust closure, we know that no captured state can ever be mutated except by the closure itself.* While this strong guarantee (which we exploit in our methodology) is not available in languages such as Java or C++, our work could nonetheless be applied *alongside* a standard formal reasoning technique for taming aliasing and side effects in the context of another programming language, such as separation logic [O'Hearn et al. 2001]. Indeed, as we will explain in Section 4, our technique can be encoded under mild assumptions about the underlying logic employed for heap reasoning.

2.4.1 What Is Special about Rust's Type System? The Rust language provides memory safety guarantees by requiring the programmer to adhere to a strict *ownership* discipline, wherein every memory-allocated value has exactly one owner, a variable, which is responsible for deallocating the value once it goes out of scope.[2] Ownership may be transferred between variables in so-called

[2]The ownership system can be temporarily disabled by manually casting to raw pointer types and operating on these; this (and similar low-level operations) must be wrapped inside so-called *unsafe* blocks. They allow for direct, untyped memory accesses (as sometimes required in systems programming), creation of cyclic data structures, etc. Using unsafe blocks shifts the responsibility for ensuring memory safety and other correctness properties from the type checker to the programmer; it is therefore understood that unsafe blocks should be used sparingly and with care, a policy that Rust programmers seem to

move assignments, and temporary references (called *borrows*) may be constructed as long as the compiler can prove that their lifetime does not exceed that of the owning variable. In addition, mutable aliasing is prohibited, meaning that mutable borrows are exclusive. These restrictions simplify reasoning about the values of memory locations because they implicitly provide *framing guarantees:* so long as a reference is live, it is guaranteed that its value will never change via aliases to the same memory, even across arbitrary function calls (and, although not essential for this paper, across different threads). The basic usage of the type system is illustrated in the following code fragment and explained in the accompanying comments:

```
   let i = Box::new(42);     // i now owns a heap-allocated integer with value 42
   let mut k = i;            // the boxed int owned by i is moved into new owner k
/* println!("i: {}", i); */  // error: use of moved value i
   let r = &k;               // borrow k immutably - both r and k can (only) read
/* *k += 1;              */  // error: values may not be modified while borrowed
/* let mut k2 = k;       */  // error: values may not be moved while borrowed
/* **r += 1;             */  // error: r is an immutable reference
   println!("*r:␣{}", *r);   // borrow expires here as r is not used afterwards
   *k += 1;                  // k is not borrowed, so modifications are allowed
   let m = &mut k;           // borrow k mutably
/* let m2 = &mut k;      */  // error: mutable borrows are exclusive
   **m += 1;                 // borrow expires here
   std::mem::drop(k);        // (only) the owner may deallocate the object
```

2.4.2 How Does Rust's Ownership System Relate to Closures? The restrictions described above also apply to closures. Arguments and return values may have value or reference types, and the captured variables are captured either by immutable or mutable borrow, or by move, depending on how the value is used within the closure, with the same framing, memory safety, and non-aliasing guarantees as above. The different modes of variable capture are illustrated in the following:

```
let a = 0;
let mut b = 0;
let c = 0;
let mut cl = || {
    println!("a:␣{}", a); // a is captured by immutable reference
    b += 1;               // b is captured by mutable reference
    std::mem::drop(c);    // c is moved into the closure
};
cl();
assert_eq!(a, 0);
assert_eq!(b, 1);
// c has been moved and cannot be accessed here anymore
```

In particular, a closure's captured state will never be modified except by calling the closure, as long as the closure instance is live. Note that this is in stark contrast to closures in languages such as Java and C++, in which aliasing is not restricted by the language.

at least partially adhere to in practice [Astrauskas et al. 2020; Evans et al. 2020]. Since Rust's philosophy is that high-level code should delegate such practices to libraries, in this paper, we only consider the safe subset of Rust.

3 METHODOLOGY

We now discuss how to write specifications for Rust closures and the rationale underlying their modular verification, thereby addressing the key problems from above.

3.1 Specifying Closure Behaviour (Key Problem K1)

As for ordinary functions, we specify a closure's behaviour using pre- and postconditions, where the postcondition may refer to argument values evaluated in the prestate (i.e. before the call) via *old expressions* [Leavens et al. 2008]. Unlike ordinary functions, though, closures may read and potentially modify variables in their captured state (→ key problem K1), which we accommodate by admitting captured variables in closure specifications.[3] For example, consider the closure specification (lines 3 and 4) in the following code snippet:

```
1  let mut a = 0;
2  let mut cl =
3      #[requires(true)]
4      #[ensures(a == old(a) + i && result == a)]
5      |i: i32| { a += i; a }; // captures a by mutable reference
6  assert_eq!(cl(1), 1);
7  assert_eq!(cl(2), 3);
```

The closure `cl` mutably captures the integer variable `a`; a call to the closure adds its argument `i` to `a`, and returns the resulting value. The given postcondition (line 4) precisely specifies this behaviour by referring to the values of `a` before (`old(a)`) and after (`a`) calling the closure; the reserved keyword `result` can be used in specifications to refer to the closure call's return value.

We say that a closure specification is *valid* if calling the closure with any arguments in any state satisfying the precondition can only lead to poststates satisfying the postcondition. The above specification for `cl` satisfies this requirement and is therefore valid. We can use this specification to prove the assertions in lines 6 and 7, i.e. we do not need to inspect the closure's body or inline the calls to reason about their results and side effects.

3.2 Specifying Evolutions of Captured State (Key Problem K2)

Pre- and postconditions are capable of relating the immediate pre- and poststates of concrete closure calls. However, many intuitively correct closure properties rely on restricting the captured state's valid evolutions across a potentially unbounded number of calls (→ key problem K2). For example, consider the following closure, which captures and increments a mutable variable `count`:

```
let mut count: i32 = 0;
let mut cl = || { count += 1; count };
```

If we ignore the possibility of integer overflows, `cl` will always return a positive result since `count` is initialised with zero and never decreased. Moreover, `count` cannot become negative due to interference from other code: since `cl` mutably captures `count`, Rust's type system prevents modifications of `count` outside of `cl`'s body until `cl` expires.

Nonetheless, the simple postcondition `result > 0` would *not* be valid because it does not hold for *all* possible prestates satisfying the implicit precondition **`true`**. To obtain a valid specification, we could strengthen both pre- and the postcondition by adding the assertion `count >= 0`; however, such a precondition would put the burden on the caller, even though it is guaranteed by the closure

[3]As a consequence, within our specifications we slightly deviate from standard Rust, which does not allow accessing a closure's captured state from outside its body (unlike, say, the fields of a struct) while the closure instance is live.

```
let i = cl();
foo(&mut cl);

let j = cl();
assert!(i < j);
```

(a) Relating two closure states across an unbounded number of calls.

```
let mut cl =
    #[invariant(count >= old(count))]
    #[ensures(count == old(count) + 1)]
    #[ensures(result == count)]
    || { count += 1; count };
```

(b) A closure with a history invariant.

Fig. 2. An example of a call site that requires tracking the evolution of a closure's captured state, and a closure declaration with a history invariant to allow such reasoning.

implementation and not the caller's responsibility. Moreover, if we pass the closure to a higher-order function, which calls the closure an unknown number of times, the knowledge that `count >= 0` would be lost because pre- and postconditions only relate two known, concrete states.

What we need is a way of expressing constraints on a closure instance's captured state that hold for *all* reachable states, even if the precise closure state is unknown after arbitrarily many calls. To this end, we introduce *invariants* into our closure specifications, which restrict the possible values that the captured variables may ever have during the entire lifetime of the closure. For example:

```
1  let mut count = 0;
2  let mut cl =
3      #[invariant(count >= 0)]
4      #[ensures(count == old(count) + 1 && result == count)]
5      #[ensures(result > 0)]
6      || { count += 1; count };
```

Our invariants are required to hold in all pre- and poststates of calls to that closure, not in intermediate states occurring during the closure's execution. Temporary violations are permitted because calls to that closure are in such a state, and Rust prevents access to the captured variables from the outside while the closure instance is live. Moreover, Rust does not support recursive closures [Matsakis 2013].

Using the above invariant, if a higher-order function calls `cl` an unknown number of times, we may no longer know the precise value of `count`, but we *will* know `count >= 0` and, therefore, that any subsequent or previous result returned by the closure must be at least positive; in other words, we can prove the validity of the postcondition on line 5.

History Invariants. Single-state invariants remain, however, insufficient to verify examples such as the one shown in Figure 2a. The assertion `i < j` requires reasoning about valid evolutions of the captured state *with respect to* a concrete earlier state; in other words, given the result of the first call to `cl`, we need a way of restricting *future* closure states reachable from this known state, again across a potentially unbounded and unknown number of calls to the closure. We know that `cl` never decreases `count` and can therefore intuitively conclude that `old(count)` must always be less than or equal to `count`, where `old(count)` refers to *any* previous closure state (including the same state reflexively). To formally express such properties, we use *history invariants:* two-state invariants that are (a) reflexive and (b) transitive.[4] Figure 2b shows a specification for `cl` that uses a history invariant that, together with the postcondition, is sufficient to prove the example in Fig. 2a.

[4]We use a refined notion of reflexivity, which takes single-state invariants into account; see Section 4.2 for the definition.

Single-state invariants can be seen as history invariants which do not constrain an earlier state, that is, do not contain `old` expressions. Therefore, we focus on history invariants in the rest of the paper, subsuming single-state invariants.

3.3 Writing Higher-Order Specifications (Key Problems K3, K2)

A higher-order function receiving a closure as an argument typically has to specify requirements on the closure's behaviour. At the very least, it will need to know that the closure's precondition holds whenever it is called. Additionally, in order to guarantee its own functional specification, a higher-order function may have expectations about the closure's side effects and results when called with specific arguments. At the higher-order function's call site, we need a way to check whether a concrete argument closure fulfils the higher-order function's requirements ($\rightarrow$ key problem K3). For this purpose, we introduce *specification entailments*, assertions of the form

$$\texttt{cl} \models |a_1, a_2, \dots| \,\{\, \texttt{requires}\left(P_{exp}\right),\ \texttt{ensures}\left(Q_{exp}\right) \,\},$$

where `cl` is the closure instance in question and a_i is the binder for the closure's i-th argument. The specification entailment guarantees that we can treat all calls to `cl` according to the *expected* specification defined by P_{exp} and Q_{exp} (as usual for postconditions, Q_{exp} may contain old-expressions and refer to `result`). This specification need *not* be identical to the *actual* specification (say, precondition $P_{\texttt{cl}}$ and postcondition $Q_{\texttt{cl}}$) given when `cl` was originally declared. Instead, the actual specification must be *at least as strong* as the expected one, according to the standard rules for behavioural subtyping [Dhara and Leavens 1996; Leavens and Naumann 2015; Liskov and Wing 1994]. That is, the specification entailment holds if (for all *future* states, as explained shortly)

$$P_{exp} \implies P_{\texttt{cl}} \quad \text{and} \quad \texttt{old}(P_{exp}) \implies \left(Q_{\texttt{cl}} \implies Q_{exp}\right),$$

where $\texttt{old}(P_{exp})$ denotes that we evaluate P_{exp} in the prestate instead of a future state.

It is this notion of subtyping on closure specifications which gives our modular technique its power, allowing us e.g. to pass many different closures (with different actual specifications) as arguments to the same higher-order function. We can use specification entailments in preconditions of higher-order functions to conveniently express requirements on the behaviour of their closure arguments, i.e. we may assume in the higher-order function's body that the argument closures fulfil *at least* the expected specifications, as given by the specification entailments in the function's precondition. Conversely, at every call site of a higher-order function, the caller has to guarantee that all passed closures fulfil their expected specification.

Similar notions that enable the substitution of one program component by another exist in other settings, for instance, behavioural subtyping, refinement theory, and higher-order program logics. However, our definition of specification entailment is novel in two major ways. First, it lends itself to automation via SMT solvers. Even though it is used for the specification of higher-order functions in the presence of mutable state, our specification entailment is defined via standard first-order implications. Existing approaches to closure verification require more complex logics or quantification over *entire* program states (to ensure the entailment holds in the state the closure is invoked). Second, specification entailments need not hold for all theoretically possible calls to a closure, but only those that may occur from the current state onwards. That is, when proving a specification entailment, one may use any knowledge about the current state of the closure's captured state and how it may evolve (as constrained by history invariants), see Section 4.3 for details. This weaker proof obligation (weakened via the history invariants) is sufficient because Rust's type system prevents modifications of the captured state via aliases.

As running example for this section, consider the higher-order function `call_ret` in Figure 3, which calls its argument closure `f` on a randomly chosen positive integer and returns the result. To

```
#[ensures(result > 0)]
fn roll_dice() -> i32 { /* ... */ }

#[requires(f |= |i: i32| { requires(i > 0), ensures(result > 0) })]
#[ensures(result > 0)]
fn call_ret(mut f: impl FnMut(i32) -> i32) -> i32 {
    let x = roll_dice();
    return f(x);
}
```

Fig. 3. Running example for reasoning about higher-order functions.

ensure that `call_ret` returns a positive integer, the specification entailment in the precondition requires that `f` maps positive integers to positive integers, i.e. `f` may be called with any positive integer and guarantees that its result, when called with such an argument, will also be positive.

The example closure `cl` below can be passed into `call_ret`, because its specification entails the expected specification in `call_ret`'s precondition: `cl` may be called *at least* with any positive integer as an argument, and it guarantees a positive result *when called with a positive integer argument.*[5] Hence, the higher-order function call in line 5 is valid:

```
1  let cl =
2      #[requires(true)]
3      #[ensures(result == i * 2)] // does not imply result > 0 by itself
4      |i: i32| -> i32 { i * 2 };
5  call_ret(cl); // still valid, because cl may assume a stronger precondition in
6                // the specification entailment in call_ret's precondition
```

Higher-order Functions and Evolving Captured State. The evolution of captured state across multiple closure calls may influence whether a specification entailment is valid and, consequently, whether a closure can be passed to a higher-order function (⟶ key problem K2). For instance, assume we wish to pass the following closure `cl` to the higher-order function `call_ret` in Figure 3:

```
1   let mut x = -1;
2   let mut cl =
3       #[invariant(x >= old(x))]
4       #[requires(i > 0)]
5       #[ensures(result == old(x) && x == old(x) + i)]
6       |i: i32| { let r = x; x += i; r };
7   // L1
8   cl(2);
9   // L2
10  call_ret(cl);
```

At position `L1`, the closure's postcondition does *not* imply `result > 0`; passing `cl` to `call_ret` would thus violate `call_ret`'s specification. However, after calling `cl` once with argument 2, we can conclude (at `L2`) from the closure's specification and the initial value of `x` that `x` is now positive. Together with the history invariant, this means that, *from now on,* all calls to `cl` will return a positive result. Hence, the higher-order function call in line 10 satisfies `call_ret`'s specification.

[5]Note that writing `i` in the closure's postcondition is equivalent to writing `old(i)`, because `i` is immutable.

Specifications for Single Calls. Many functions in the Rust standard library, including, for instance, `Option::map()`, `Result::and_then()`, and `hash_map::Entry::or_insert_with()`, call their argument closure(s) at most once. Rust even has a special trait, `FnOnce`, that a higher-order function can use as a generic trait bound to express that it will call the closure at most once.

As a consequence, the specification entailment operator that we have discussed so far is too imprecise for such situations, because it reasons about a closure's state after an arbitrary number of calls have been made. We therefore introduce a specialised *single-call specification entailment* `|=!`, expressing that the closure instance on its left-hand side fulfils the expected specification *for its next call.* For example, if we change the precondition of `call_ret` in Figure 3 to use `|=!` instead of `|=`, we can pass in the following closure, which returns a positive value only when first called:

```
let mut x = 1;
let mut cl =
    #[requires(i > 0)]
    #[ensures(result == old(x) && x == old(x) - i)]
    |i: i32| { let r = x; x -= i; r };
```

3.4 Describing Effects of Closure Calls (Key Problem K4)

Describing a higher-order function's behaviour and effects may rely on the fact that its closure argument(s) have themselves been called with certain arguments, such as all elements in a collection, and on the results and side effects of such calls (e.g. that every element in `map()`'s result is the result of having called the argument closure with an element from the input collection).

However, such properties *cannot* be expressed using the specification constructs that we have presented so far, which provide only guarantees about potential future calls. To specify the above properties in a first-order setting, we need a way of post hoc expressing the effects of closure calls made by a higher-order function in a way that is parametric in the behaviour of the closure (→ key problem K4). To this end, we propose a novel specification construct describing *opaquely* that concrete closure calls have happened, i.e. without knowing what the closure does exactly (which is generally necessary for specifying any generic higher-order function). A *call description*

$$\texttt{cl}\ (a_1, a_2, \dots) \rightsquigarrow \{Q\}$$

specifies that "`cl` was called with concrete arguments $a_1, a_2, \dots$, such that the assertion Q held in its poststate". Here, `cl` and $a_1, a_2, \dots$ are evaluated in the *current* state, i.e. the higher-order function's poststate if the call description occurs in the postcondition of a higher-order function. For instance, consider the following specification for our running example `call_ret` (Figure 3):

```
#[requires(f |= |i: i32| { requires(i > 0), ensures(true) })]
#[ensures(exists(|i: i32| i > 0 && old(f) (i) ~~> { result == outer(result) }))]
```

The above call description expresses that the closure `f` has been called with some (existentially quantified) integer `i` as an argument, and that its result was equal to the value returned by `call_ret`. Notice that, like specification entailments, the call description is a *nested* specification whose "inner" pre- and poststate (which belong to the closure call) are different from the "outer" pre- and poststate (which belong to the higher-order function). To disambiguate these states, we wrap expressions in `outer()` if we have to access the outer pre-/poststate from within the call description, similar to `old()` expressions. Furthermore, we use `old(f)` on the call description's left-hand side since `f`'s captured state might change during every call (i.e. the pre- and poststate values of `f`, which include its captured state, may be different); therefore, the old-expression enables us to refer to the captured state in which we have called the closure instance.

```rust
#[requires(f |= |arg: &mut T| { requires(outer(v).contains(arg)) })]
#[ensures(v.len() == old(v.len()))]
fn map<T, U>(v: &mut Vec<T>, mut f: impl FnMut(&mut T) -> U) -> Vec<U> {
    let mut r = vec![];
    for el in v {
        r.push(f(el));
    }
    r
}
```

Fig. 4. Simplified version of the standard library function `map()`, with a possible specification.

Although the call descriptions by themselves do not convey enough information to precisely describe what the closure call has actually computed (e.g. we may not know the concrete types of all arguments), they can be combined at the higher-order function's call site with the specification of the concrete closure that was passed as an argument to the higher-order function. For instance, continuing our running example, the following assertion in line 6 is valid given the specifications of `call_ret` and `cl`: `cl`'s specification tells us that its result will always be greater than 42 and even, and `call_ret`'s specification tells us that the higher-order function's result was computed by calling `cl` with some positive argument. Hence, the assertion holds.

```rust
1  let cl =
2      #[requires(i > 0)]
3      #[ensures(result > 42 && result % 2 == 0)]
4      |i: i32| -> i32 { 42 + 2 * i };
5  let r = call_ret(cl);
6  assert!(r > 42 && r % 2 == 0);
```

Unbounded Closure Calls. In the above example, closures were called only a statically bounded number of times; we now turn to a more interesting example where the number of closure calls is unbounded. Figure 4 shows a (simplified) implementation of the higher-order function `map()` from Rust's standard library that calls its argument closure `f` once for every element in a vector `v`. Its precondition specifies that `f` may assume that its argument is contained in `v` (i.e. it has any property provable for all elements in `v` at `map`'s call site). Its postcondition states only that `map()`'s result will have the same length as the input vector. `f` receives a mutable reference and produces a result; we want to describe both the result and the side effects on the arguments in `map`'s specification.

First, note that `f` implements `FnMut` and thus may have its own mutable captured state. Moreover, since `f` is called on each vector element in a loop, we cannot put an a priori bound on the number of calls to `f`. Therefore, we cannot specify the precise closure instance used in every call. Instead, we propose a relaxed notation for closure instances in call descriptions: we write `:f` (read: "some `f`") to denote that we did not necessarily call the exact closure instance `f` but *some* intermediate closure instance $\hat{\mathtt{f}}$ (i.e. the same closure as `f` but with a potentially different version of captured state) such that the history invariants hold between `old(f)` and $\hat{\mathtt{f}}$ as well as between $\hat{\mathtt{f}}$ and `f`: in other words, that we called an in-between closure instance (between those in `map`'s pre- and poststates).

Second, since `f` receives a mutable reference, it is insufficient to describe which reference `f` was called with; we also need to describe what the contents of the reference looked like in `f`'s pre- and poststate. To this end, we propose to use the colon notation for arguments as well and extend our call description syntax with an optional prestate description P:

$$[:]\texttt{cl}\ ([:]a_1, [:]a_2, \dots)\ \{P\} \rightsquigarrow \{Q\}$$

Here, $:\ a_i$ introduces a binder for the argument, which can be evaluated in both P and Q to describe the argument value in both the closure call's pre- and poststate. Put together, we can express the desired specification of the `map()` function, that is, `map()` returns a vector obtained from applying its argument closure `f` pointwise to input vector `v`, by adding the following postcondition to Figure 4:

```
#[ensures(forall(|i: usize|
    i < v.len() ==>
        :f (:arg) { *arg == outer(old(v[i])) } ~~>
            { *arg == outer(v[i]) && result == outer(result[i]) }
))]
```

More precisely, the above postcondition quantifies over all positions i in the input vector `v` and uses a call description to specify that `f` is called for each such i. The prestate description requires that the i-th call of `f` takes the i-th element of `v` as an argument; the poststate description ensures that the i-th call of `f` produces the i-th element of the vector returned by `map()`, i.e. `outer(result[i])`.

3.5 Ghost Functions as Abstract Predicates (Key Problems K5, K6)

For specifying the behaviour of higher-order functions performing complex aggregations such as `fold()` (→ key problem K5), it is insufficient to describe individual closure calls; instead, we have to establish and maintain properties of the intermediate results that will eventually lead up to the overall result. This issue is not specific to reasoning about closures and higher-order functions: an analogous problem arises when verifying loops, where *loop invariants* provide an elegant mechanism for specifying properties about intermediate results. Some invariants are naturally expressed using quantifiers: in a `find` method, we may state that no element up to some index equals the key. Other invariants are more easily expressed using sequence comprehensions or ghost code: in a `sum` method, a ghost variable may store the sum up to some position in an array.

To represent such invariants for reasoning about higher-order functions, we propose passing pure, i.e. terminating and side-effect-free, functions returning booleans as *ghost arguments* to represent the invariants. Ghost arguments per se are also not specific to reasoning about closures or higher-order functions and can be viewed as an orthogonal extension of our methodology. However, they allow us to enrich higher-order specifications in important ways by replacing higher-order quantifications over assertions (here, the invariants) by explicit ghost state, which is crucial for automated reasoning about closure specifications.

To illustrate the intended usage of ghost arguments for specifying aggregation functions, consider the higher-order function `fold_vec` in Figure 5, which computes an accumulated value – stored in variable `acc` – by applying its closure argument `f` to every element in a vector. We use a ghost argument function `inv` that takes a vector, an integer, and a value of the accumulation type to represent the invariant that `fold_vec` maintains on the accumulated value. More specifically, `inv(v, i, a)` expresses that `a` accumulates the first `i` elements of `v`. The specification requires that:

(1) the invariant holds for the initial value of `acc` when no vector elements have been processed yet (line 2); and
(2) the invariant is preserved by calling the closure, i.e. we use a specification entailment expressing that the closure may assume the invariant for the first `n` elements of the vector (line 5) and that its second argument, the element currently being added to the accumulation, is in the vector (line 6); the specification entailment's postcondition has to guarantee that, after calling the closure, the invariant will hold for the first `n + 1` elements of `v`.

```
#[ghost_arg(inv: Fn(&Vec<T>, usize, A) -> bool)]
#[requires(inv(v, 0, init))]
#[requires(forall(|n: usize|
    n < v.len() ==> // outer(v) has been introduced in Sec. 3.4
        f |= |a, c| { requires(inv(outer(v), outer(n), a)
                          && outer(v[n]) == c),
                      ensures(inv(outer(v), outer(n) + 1, result)) } ))]
#[ensures(inv(v, v.len(), result))]
fn fold_vec<T, A>(v: &Vec<T>, init: A, mut f: impl FnMut(A, &T) -> A) -> A {
    let mut acc = init;
    for el in v {
        acc = f(acc, el);
    }
    acc
}
```

Fig. 5. A ghost predicate specifying an invariant for a folding operation on a vector.

After having folded all elements, `fold_vec` can guarantee that the invariant holds for the entire input vector and its own result, the final accumulated value. The caller has to pick `inv` so that it matches the closure's behaviour, but `fold_vec` makes no further assumptions about it, allowing for maximal flexibility in the kind of accumulations performed with this function.

The following example shows how `fold_vec` is called on a vector `v`; the concrete invariant `inv(v, idx, acc)` is supplied as a ghost argument in line 5: it states that the accumulated value `acc` equals the sum of the vector's first `idx` elements, which complies with `fold_vec`'s precondition.[6]

```
1  let v: Vec<i32> = vec![1,2,3];
2  let sum: i32 = fold_vec(
3      &v, 0,
4      #[ensures(result == x + y)] |x: i32, y: i32| x + y,
5      #[ghost] |v: Vec<i32>, idx: usize, acc: i32| v.to_seq()[0..idx].sum() == acc);
6  assert!(v == 6);
```

Closures as Predicates. We also propose the use of ghost argument functions for higher-order functions where the closure represents some mathematical predicate (→ key problem K6). For instance, the `sort_by()` function sorts a vector according to the ordering relation implemented by its argument closure. We can express requirements such as reflexivity, transitivity, and antisymmetry, which are required to form an ordering relation, as well as the sortedness property of the result, using a ghost argument function returning a boolean, which we relate to the (potentially impure) behaviour of the closure. For example, reflexivity can be expressed as follows:

$$\texttt{cl} \;\models\; |\,\mathsf{a},\, \mathsf{b}\,|\; \{\, \mathsf{ensures}\,(\mathsf{a} == \mathsf{b} \implies \texttt{result})\; \}\,,$$

For transitivity, we would consider specification entailments across triples of future closure states. The advantage of pure ghost argument functions is that they can be used much more freely in specifications than the closure itself. For example, we can call a ghost argument function, but not the closure, in the specification, which allows for much richer specifications whenever non-trivial

[6]The notation `v.to_seq()[0..idx].sum()` first converts the vector to a mathematical sequence consisting of the first `idx` elements and then computes their sum. Sequence types are widely supported by intermediate verification languages.

```
#[requires(fc |= |i| { requires(i >= 0), ensures(result >= 0) })]
#[ensures(:fc (:i) { i == 5 } ~~> { true })]
#[ensures(result >= 0)]
fn hof<T: FnMut(i32) -> i32>(fc: &mut T) -> i32 { fc(5) }

fn main() {
  let mut x = 0;
  let mut cl =
    #[requires(i >= 0)]
    #[invariant(x >= old(x))]
    #[ensures(result >= 0)]
    |i: i32| -> i32 { x += i; i };
  hof(&mut cl);
}
```

Fig. 6. Running example illustrating our first-order encoding.

mathematical properties need to be expressed. However, many simple functions where closures are used as predicates (⟶ Section 2.3.4), such as `filter()`, `any()`, and `all()`, can also be specified using call descriptions, removing the need for explicitly passing a ghost argument at every call site.

The specification constructs presented in this section address all of the key problems identified in Section 2 and allow us to express precise, modular specifications of the common use cases of closures in Rust. Next, we will explain how we verify Rust implementations against such specifications.

4 FIRST-ORDER ENCODING

Automated deductive verification tools such as Boogie [Barnett et al. 2006], Dafny [Leino 2010], VeriFast [Jacobs et al. 2011], and Viper [Müller et al. 2016] ultimately reduce verification to checking satisfiability of first-order logic formulas modulo suitable theories. However, our methodology for reasoning about Rust closures heavily relies on *higher-order* concepts, e.g. nested closure specifications (specification entailments and call descriptions). These concepts have no counterpart in first-order logic and are not directly supported by the above tools.

In this section, we develop an encoding strategy that breaks down these higher-order concepts into first-order logic, which allows integrating our methodology into automated verifiers. We present our encoding strategy in terms of a simple first-order logic that is supported by all of the above verification tools; our concrete implementation builds on Prusti [Astrauskas et al. 2019] which, in turn, uses the Viper verification framework.

Running Example. Throughout this section, we consider the Rust program in Figure 6. The higher-order function `hof` takes a closure of type `T: FnMut(i32) -> i32` (i.e. a closure with potentially mutable captured state which takes and returns an integer) and is equipped with a specification entailment, which states that non-negative integers are mapped to non-negative integers, in its precondition and a call description, which states that the closure is called with argument 5, in its postcondition. The function `main` creates a closure instance `cl`, which mutably captures the variable `x`, and calls `hof` with a mutable reference to `cl`. We will present the most relevant aspects of the above example's encoding; its full encoding is available online [Wolff et al. 2021b].

4.1 Prerequisites

We assume a first-order logic that is supported by SMT solvers and in which we can define and axiomatise uninterpreted sorts and functions. Moreover, we impose the following two requirements:

1. Framing. We need an encoding of the program memory together with a methodology for *framing* information around calls We rely on framing since our encoding reasons only about those parts of the memory that are directly affected by closure calls, i.e. a closure's arguments and its captured state. Framing guarantees that such calls do not invalidate our knowledge about the rest of the program state. Existing automated verifiers support framing via well-established methodologies for instance, by using separation logic in VeriFast and Viper, and dynamic frames in Dafny.

2. Snapshots. We need a mechanism *snap* that takes a (possibly heap-dependent) value in the current program memory and produces a mathematical abstraction (its "snapshot" [Smans et al. 2010]) that uniquely characterises the value such that it can be passed around, copied, compared with snapshots of other values (even in other states), and quantified over. We rely on snapshots for reasoning about possible future (closure) states without resorting to higher-order concepts.

For instance, the snapshots of a data type, say **`struct`** `S { a: i32, b: i32 }`, can be modelled as an uninterpreted sort $\texttt{Snap}_{\texttt{S}}$, equipped with uninterpreted functions $\texttt{cons}_{\texttt{S}}\texttt{(i32, i32) -> Snap}_{\texttt{S}}$, $\texttt{a}_{\texttt{S}}\texttt{(Snap}_{\texttt{S}}\texttt{) -> i32}$, and $\texttt{b}_{\texttt{S}}\texttt{(Snap}_{\texttt{S}}\texttt{) -> i32}$, which mimic the struct's (injective) constructor and getter functions for accessing field values, respectively. Computing the snapshot of a struct from the program memory (*snap*) then amounts to calling its constructor with the snapshot values of its fields.[7] Conversely, to convert a snapshot s back to a struct, we take a fresh struct instance and *assume* that its snapshot (as computed by *snap*) is equal to s. We will not distinguish between heap objects and their snapshots throughout this section.

Taking snapshots of *closure instances*, i.e. computing mathematical abstractions of their captured state, works analogously because we can view closure instances as structs with a field for each captured variable. Existing automated verifiers enable the use of snapshots, typically by providing a way to declare and axiomatise uninterpreted functions; some (e.g. VeriFast [Jacobs et al. 2011]) employ the same concept in their underlying encodings.

Running Example (Cont.). We can view the closure `cl` as an instance of **`struct`** `CL { x: i32 }`; *snap* (`cl`) yields an element of the uninterpreted sort $\texttt{Snap}_{\texttt{CL}}$ satisfying $\texttt{x}_{\texttt{CL}}(\textit{snap}(\texttt{cl})) == \texttt{x}$.

4.2 Modelling Closure Specifications

Recall that every closure specification consists of a precondition, a history invariant, and a postcondition. However, relating a concrete closure instance to its specification is non-trivial: in the context of modular verification, we cannot, in general, know from which closure definition a concrete closure instance originates. Thus, we do not usually know the declared, most precise specification, but only, perhaps multiple, *weakened* specifications (cf. Section 3.3). Instead of attempting to manually keep track of all known specifications for a given closure instance, we represent the actual closure specification by uninterpreted *specification functions* and let the SMT solver figure out the details; a similar approach has been proposed by Nordio et al. [2010] and Kassios and Müller [2010]. More precisely, for every closure definition and generic closure type parameter, we introduce three

[7] We identify Rust's integer types with the type `Int` in our underlying logic and therefore omit `snap(...)` for integers.

uninterpreted functions, modelling the closure's precondition, postcondition, and history invariant:

$$\begin{aligned} \mathit{pre}\,(c, a_1, a_2, \dots) &: \texttt{Bool} \\ \mathit{post}\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \dots, r) &: \texttt{Bool} \\ \mathit{inv}\,(\overline{c}, c) &: \texttt{Bool} \end{aligned}$$

In the above functions, c refers to a closure instance, a_i to the closure's i-th argument, and r to its return value. Moreover, $\overline{c}$ and $\overline{a_i}$ refer to old values, i.e. the value in the prestate for *post* and the value in any former (or the current) state for *inv*. All specification functions are mathematical functions independent of program states; closure instances (including their captured state) and heap-dependent arguments are passed as snapshots. Our encoding will ensure that a specification function evaluates to true only if the corresponding precondition (resp. postcondition or invariant) holds for the given closure instance, arguments, and return value in the given program states.

For the function *inv*, we may assume transitivity and reflexivity modulo the single-state invariants (i.e. $\mathit{inv}(c, c)$ only holds if c satisfies the single-state invariants) as an axiom; this is justified by proving both properties separately for any concrete closure declaration. In particular, the following formula characterises reflexivity modulo single-state invariants [Cohen et al. 2015]:

$$\forall c\,.\ (\exists c'\,.\ \mathit{inv}\,(c', c)) \implies \mathit{inv}\,(c, c)$$

where c and c' range over closure instances. Intuitively, the left-hand side of the implication above filters out exactly those instances c from the universal quantification for which the single-state invariants hold.[8] In particular, $\mathit{inv}\,(c', c)$ asserts that precisely the single-state invariants hold for closure instance c.

Running Example (Cont.). We generate two sets of specification functions, one for the concrete closure type `CL` of `cl`, and one for the generic closure type `T` passed to `hof`. Both sets of specification functions share the same signature save for the snapshot type, which models a closure's captured state (x for `CL` and arbitrary state for `T`). Formally, for $\texttt{S} \in \{\texttt{CL}, \texttt{T}\}$, the specification functions are:

$$\begin{aligned} \mathit{pre}_{\texttt{S}}\,(c : \texttt{Snap}_{\texttt{S}},\ \ i : \texttt{Int}) &: \texttt{Bool} \\ \mathit{post}_{\texttt{S}}\left(\overline{c} : \texttt{Snap}_{\texttt{S}},\ \ c : \texttt{Snap}_{\texttt{S}},\ \ \overline{i} : \texttt{Int},\ \ i : \texttt{Int},\ \ \mathit{result} : \texttt{Int}\right) &: \texttt{Bool} \\ \mathit{inv}_{\texttt{S}}\,(\overline{c} : \texttt{Snap}_{\texttt{S}},\ \ c : \texttt{Snap}_{\texttt{S}}) &: \texttt{Bool} \end{aligned}$$

Specification functions provide a means of symbolically denoting the individual components of a closure instance's specification without precisely knowing the specification or closure definition. In the remainder of this section, we show how we use these functions to encode closures and our various higher-order specification features for reasoning about them.

4.3 Breaking Down Higher-Order Specifications

We now show how to encode usages of the two higher-order specification constructs proposed in Section 3 (specification entailments and call descriptions) into first-order logic.

Encoding Specification Entailments. Recall from Section 3.3 that a specification entailment

$$\texttt{cl} \models |a_1, a_2, \dots|\ \{\texttt{requires}\,(P)\,, \texttt{ensures}\,(Q)\}$$

expresses that the closure `cl`'s specification is stronger (in the sense of behavioural subtyping) than the expected specification given by precondition P and postcondition Q for all valid (w.r.t. to the

[8]In our explanations, we assume *satisfiability* of the history invariants: An invariant of the form $\mathit{old}\,(x) < x \land \mathit{old}\,(x) > x$ is allowed but always false, regardless of the concrete old and new states of x. However, we then either cannot prove the closure body, or the closure does not terminate (and thus trivially verifies in a partial correctness setting).

closure's invariants) and reachable states of `cl`'s captured state; in other words, that `cl` satisfies the expected specification for all future calls made to it.

Formally, if *cur* is the closure instance `cl` in its current state, this means that (1) for all possible future states $\overline{c}$ of `cl` – i.e., those satisfying $inv\,(cur, \overline{c})$ – P implies `cl`'s actual precondition $pre\,(\ldots)$, and (2) whenever the expected precondition P holds for a valid future state $\overline{c}$ of `cl` then the actual postcondition $post\,(\ldots)$ implies the expected postcondition Q:[9]

$$\forall\, \overline{c}, \overline{a_1}, \overline{a_2}, \ldots\, .\ inv\,(cur, \overline{c}) \implies (P\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots) \implies pre\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots)) \tag{1}$$

$$\begin{aligned}\forall\, \overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r\, .\ inv\,(cur, \overline{c}) &\implies (P\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots) \implies \\ ((post\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r) \wedge inv\,(\overline{c}, c)) &\implies Q\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r)))\end{aligned} \tag{2}$$

Running Example (Cont.). To encode the specification entailment in `hof`'s precondition, namely

$$\texttt{fc} \models |i|\,\{\texttt{requires}(i \geq 0),\ \texttt{ensures}(result \geq 0)\}\,,$$

we use the specification functions for the generic type parameter `T`:

$$\forall\, \overline{f} : \mathsf{Snap}_\mathsf{T},\ \overline{i} : \texttt{Int}\, .\ inv_\mathsf{T}(snap\,(\texttt{fc}),\, \overline{f}) \implies \left(\overline{i} \geq 0 \implies pre_\mathsf{T}(\overline{f},\, \overline{i})\right)$$

$$\begin{aligned}&\forall\, \overline{f}, f : \mathsf{Snap}_\mathsf{T},\ \overline{i}, i, result : \texttt{Int}\, .\ inv_\mathsf{T}(snap\,(\texttt{fc}),\, \overline{f}) \implies \\ &\quad \left(\overline{i} \geq 0 \implies \left(\left(post_\mathsf{T}(\overline{f},\, f,\, \overline{i},\, i,\, result) \wedge inv_\mathsf{T}(\overline{f},\, f)\right) \implies result \geq 0\right)\right)\end{aligned}$$

Encoding Call Descriptions. Recall from Section 3.4 that a call description of the form

$$:\texttt{f}\,(:a_1, :a_2, \ldots)\ \{P\} \rightsquigarrow \{Q\}$$

conceptually describes a specific closure call that happened in the past. More precisely, `:f` refers to *some* closure instance (with possibly modified captured state) "between" `old(f)` and `f`, i.e. an existentially quantified instance $\overline{c}$ satisfying $inv\,(\texttt{old}\,(f), \overline{c})$ that was called to produce poststate instance c with $inv\,(c, f)$. Furthermore, both $pre\,(\ldots)$ and P held in the prestate of that call, and, analogously, both $post\,(\ldots)$ and Q held in its poststate. Finally, we know that the call has preserved the closure's invariants: $inv\,(\overline{c}, c)$. Putting everything together, we obtain the following first-order encoding of call descriptions:[10]

$$\begin{aligned}&\exists\, \overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r\, .\ inv\,(\texttt{old}\,(f), \overline{c}) \wedge inv\,(\overline{c}, c) \wedge inv\,(c, f) \wedge pre\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots) \\ &\quad \wedge P\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots) \wedge post\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r) \wedge Q\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r)\end{aligned}$$

Closure instances and arguments that are *not* preceded by a colon are encoded more-simply: they are not existentially quantified; we take their values from the current environment.

Running Example (Cont.). To encode the call description in `hof`'s postcondition, namely

$$:\texttt{fc}\,(:i)\ \{i == 5\} \rightsquigarrow \{\texttt{true}\}\,,$$

we again only need the specification functions for closure instances of the generic type `T`:

$$\begin{aligned}&\exists \overline{f}, f : \mathsf{Snap}_\mathsf{T},\ \overline{i}, i, result : \texttt{Int}\, . \\ &\quad inv_\mathsf{T}(\texttt{old}(snap(\texttt{fc})), \overline{f}) \wedge inv_\mathsf{T}(\overline{f}, f) \wedge inv_\mathsf{T}(f, snap\,(\texttt{fc})) \\ &\quad \wedge pre_\mathsf{T}(\overline{f}, \overline{i}) \wedge \overline{i} == 5 \wedge post_\mathsf{T}(\overline{f}, f, \overline{i}, i, result) \wedge \texttt{true}\end{aligned}$$

[9]Automating verification requires handling some SMT-specific details. Assuming the SMT solver of choice uses E-matching [de Moura and Bjørner 2007], we also have to supply matching patterns (so-called triggers) to guide quantifier instantiations. In the encoding of specification entailments, suitable triggers are $\{pre\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots)\}$ and $\{post\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r)\}$.
[10]Again, we may have to choose a suitable trigger, in this case: $\{pre\,(\overline{c}, \overline{a_1}, \overline{a_2}, \ldots), post\,(\overline{c}, c, \overline{a_1}, a_1, \overline{a_2}, a_2, \ldots, r)\}$.

4.4 Encoding Closure Calls, Instantiations, and Definitions

It remains to encode the Rust statements involving closures, i.e. definitions, instantiations, and calls. Higher-order functions receive an implicit extra precondition $inv\,(c, c)$ (i.e. that c satisfies all single-state invariants) and postcondition $inv\,(old\,(c)\,, c)$ (i.e. that the closure instance's invariants have been maintained) for each closure argument c; otherwise, they need no special treatment.

Encoding Closure Calls. The main steps of encoding a closure call are the same as for ordinary function calls: We first assert the closure's precondition $pre\,(c, a_1, a_2, \dots)$, then havoc (i.e. give up our knowledge of, according to the heap framing methodology from the underlying tool) the parts of the program memory that might have been modified during the call, namely the closure's captured state and its mutable arguments (plus everything reachable from them), and finally assume the closure's postcondition $post\,(\overline{c}, c, \overline{a_1}, a_1, \dots, r)$. Additionally, in the prestate, we assert $inv\,(c, c)$ to check the single-state invariants for c's captured state, and we may assume $inv\,(\overline{c}, c)$ in the poststate (i.e., that the call has maintained all invariants). [11]

Running Example (Cont.). For $s = snap(\texttt{fc})$, we encode the call `fc(5)` in line 4 of Figure 6 in three steps: first, we assert the closure's precondition $pre_{\top}\,(s, 5)$ and single-state invariant $inv_{\top}\,(s, s)$; then, we havoc `fc`'s captured state, and finally, we assume the closure's postcondition and invariant as

$$post_{\top}\,(\texttt{old}(s), s, \texttt{old}(5), 5) \land inv_{\top}\,(\texttt{old}(s), s)\ .$$

Encoding the higher-order function call `hof(&mut cl)` then amounts to (1) asserting its precondition, consisting of $inv_{\texttt{CL}}\,(snap(\texttt{cl}), snap(\texttt{cl}))$ and the specification entailment

$$\texttt{cl}\ \models\ |i|\ \{\ \texttt{requires}\,(i \geq 0)\,,\ \texttt{ensures}\,(result \geq 0)\ \}\,,$$

(2) havocking the state potentially modified by `hof`, and (3) assuming its postcondition, namely $\texttt{result} \geq 0$, $inv_{\texttt{CL}}\,(\texttt{old}\,(snap\,(\texttt{cl})),\ snap\,(\texttt{cl}))$, and the call description

$$\texttt{cl}\,(\texttt{i})\ \{\ \texttt{i == 5}\ \} \rightsquigarrow \{\ \texttt{true}\ \}\ .$$

Encoding Closure Definitions. Closure definitions are encoded similarly to ordinary function declarations: We assume the closure's precondition and invariants, followed by the encoding of its body, and then assert its postcondition and invariant preservation w.r.t. the prestate. Furthermore, we check the closure's single-state invariants. As already discussed in Section 4.2, they must be checked to be transitive and reflexive (modulo the single-state invariants).

Encoding Closure Instantiations. When creating a new closure instance cur, our encoding needs to check that the initial values of the captured variables satisfy the closure's declared invariants I. Since the closure definition is always known when instantiating a closure instance, we can simply assert I directly and assume $inv\,(cur, cur)$. Notice that this assumption is justified by asserting the declared invariants I but does not follow automatically because inv is an uninterpreted function. For the same reason, we have to model the correspondence between the specification functions pre and $post$ and the closure's actual (declared) specification; that is, we assume that the following specification entailment is true for the created closure instance cur (where P and Q are the pre- and postcondition as declared for the closure definition that we are instantiating):

$$cur\ \models\ |a_1, a_2, \dots|\ \{\ \texttt{requires}\,(P)\,,\ \texttt{ensures}\,(Q)\ \}$$

Finally, we relate the actual declared invariants I to the specification function inv:

$$\forall\, \overline{c}, c\ .\ inv\,(cur, \overline{c}) \implies (inv\,(\overline{c}, c) \implies I\,(\overline{c}, c))$$

[11]The asserted invariant is provable in our methodology, i.e. it cannot fail. If an invariant is violated, we will encounter an assertion failure in the encoding of closure definitions.

Running Example (Cont.). The closure `cl` (with snapshot $s = \mathit{snap}(\texttt{cl})$) is defined and instantiated in lines 8–12 of Figure 6. The closure body is encoded like a regular function body, except for the fact that it may assume the single-state invariants and must prove invariant preservation (`x >= old(x)` here). For the closure *instantiation,* we check that the invariants on the captured state are established (`x >= x`) and assume $\mathit{inv}_{\mathsf{CL}}(s, s)$. After that, we relate the declared specification to the uninterpreted specification functions by assuming the specification entailment

$$\texttt{cl} \models |\,i\,|\,\{\,\texttt{requires}\,(i \geq 0)\,,\ \texttt{ensures}\,(\mathit{result} \geq 0)\,\}\,,$$

which is encoded as before. Moreover, we relate the invariant of `cl` to its specification function:

$$\forall\,\overline{f}, f : \mathsf{Snap}_{\mathsf{CL}}\ .\ \mathit{inv}_{\mathsf{CL}}(s, \overline{f}) \implies \Big(\mathit{inv}_{\mathsf{CL}}(\overline{f}, f) \implies x_{\mathsf{CL}}(f) \geq x_{\mathsf{CL}}(\overline{f})\Big)$$

5 IMPLEMENTATION AND EVALUATION

We integrated our methodology for verifying closures and higher-order functions into the automated Rust verifier Prusti [Astrauskas et al. 2019]. Section 5.1 gives an overview of our implementation. To evaluate both our methodology and its prototypical implementation, we specified and automatically verified a number of realistic, non-trivial case studies of Rust programs involving closures. Section 5.2 presents the results of our evaluation as well as how we selected the case studies. Finally, Section 5.3 gives some justifications for our choice of common use cases in Section 2.3.

5.1 Implementation Overview

We implemented our methodology on top of the automated Rust verification tool Prusti [Astrauskas et al. 2019], which builds on the Viper verification infrastructure [Müller et al. 2016]. Our source code, its documentation, and all examples are available online [Wolff et al. 2021a,b]. Given a Rust program without any additional specification annotations, Prusti utilises information from the Rust compiler, especially its type and borrow checkers, to automatically derive a *core proof* – a memory safety proof in a permission-based separation logic formalised in Viper's intermediate verification language.[12] Additionally, Prusti allows users to annotate programs with functional specifications, such as pre- and postconditions for functions as well as loop invariants, in a subset of Rust's expression syntax (extended by quantifiers). These functional specifications are then verified on top of the core proof, which provides the basis for successful verification of the functional specification e.g. by giving framing guarantees. We employed the same principles when implementing our methodology for verifying closures and higher-order functions.

Our Prusti extension implements the encoding strategy developed in Section 4. We were able to reuse significant parts of Prusti's existing infrastructure; for example, the Rust compiler transforms closures into functions with an implicit extra argument for the closure's captured state, which is an aggregate type similar to regular structs. Hence, we were able to adapt Prusti's type encoding for structs to also work for closure types.

Our implementation supports the specification constructs presented throughout the paper with some syntactic differences. Most of these differences are superficial, e.g. the implemented version of call descriptions looks similar to specification entailments to allow us to reuse parsing code. Other differences are necessitated by limited APIs for the internals of the Rust compiler – closure declarations must be wrapped in `closure!(...)` and the programmer must declare any variable captured by the closure to be able to use it in specifications. A comparison between our idealised specification syntax and Prusti's syntax, is found in the online documentation [Wolff et al. 2021a].

[12] The borrow checker enforces adherence to the ownership system and various additional restrictions, e.g. that values may not be moved while borrowed, accessed while immutably borrowed, etc.

Before our extension, Prusti had no generally applicable mechanism for specifying the effects of closure calls and their side effects in a modular fashion. Moreover, there was also no straightforward way to write an assertion expressing side-effectful computations of closures that mutate the heap.

5.2 Experimental Evaluation

For our experimental evaluation, we verified a collection of Rust programs that demonstrate the specification and verification capabilities of our methodology.

Selection of Case Studies. We evaluated our implementation on two sets of benchmarks: first, we collected examples involving closures and higher-order functions from the verification literature. Since our tool is, to the best of our knowledge, the first supporting automated verification of Rust closures, these examples were originally written in different languages, such as C# and Scheme. We translated each of them (and their specifications) to annotated Rust programs.[13]

Second, we added our own examples to demonstrate how our verification methodology performs on concrete instances of the use cases of Rust closures identified in Section 2.3. These examples are mainly based on widely used standard library functions, such as `map()`, `fold()`, `any()`, and `all()`. Since the standard library relies on some Rust features that are both unrelated to closures and unsupported in Prusti, such as lazy iterator chaining, our example programs use custom implementations of the above functions that operate directly on vectors.[14] However, they still showcase the same challenges for closure and higher-order function verification.

An overview of all examples is presented in Table 1. It includes 10 correct examples and 4 erroneous examples, which were obtained by manually seeding errors in 4 of the correct examples. Our suite contains at least one example, including both call sites and higher-order function definitions, for every use case identified in Section 2.3. We used these to assess whether our methodology is flexible and powerful enough to specify and verify practically relevant idioms involving closures and higher-order functions; we will further justify these choices in Section 5.3.

Experimental Setup. All experiments were performed using an Intel Core i9-10885H 2.40GHz CPU with 16 GiB of RAM. We measure the wall-clock runtimes for computing the Rust-to-Viper encoding and its subsequent verification using Viper's symbolic execution backend.

Results. Table 1 shows the results of our evaluation. Our implementation reported the correct results on all 14 examples, each in less than 13s. For each example, we counted the lines of code (LOC), lines of specification (LOS), and lines of glue code (LOG), i.e. code that is solely required by Prusti to deal with currently unsupported Rust features. Overall, the annotation overhead amounts to roughly one line of specification per line of code (excluding glue code), where we performed a conservative count based on Prusti's more verbose syntax without compressing specifications, e.g. conjoining postconditions or removing whitespace. The specification effort based on the idealised syntax proposed in this paper would be significantly lower. The overall annotation overhead is relatively low – at least compared to other verifiers for imperative code. In particular, leveraging Rust's type information frees us from the burden of specifying the shape of heap structures or framing information required by other tools.

[13]While we attempted to translate each example into Rust as closely as possible, some adaptations were required in order to make the translated program comply with Rust's rigid type and ownership system.

[14]Rust's `Iterator` trait is lazy; functions such as `map()` and `filter()` only return custom `Iterator` implementations, which then perform mapping/filtering once an element is requested. Furthermore, it is common in Rust to chain calls to iterator functions. The verification of Rust's `Iterator` functions therefore requires threading any knowledge about the original values through arbitrarily many, possibly dynamically dispatched, higher-order function calls of `Iterator` and its specialised implementations. We focus on the challenges of verifying closures and higher-order functions that directly work on data structures, but without the added complexities of lazy iterator chaining.

Table 1. The list of example programs, together with the features of our methodology that they showcase, on which we have evaluated our implementation. The examples named `*_err` are negative tests expected to yield verification errors. The features are usage of any (A) or mutable (M) captured state (CS), possibly with history invariants (HI), as well as higher-order specifications (HO) using specification entailments (SE), call descriptions (CD), or ghost argument functions (GA). Column NC shows whether the number of closure calls in the given example is bounded (=) or unbounded (≥); e.g. a `map()` on a vector would qualify as "unboundedly many" calls, even if the example includes a call site with a fixed number of elements in the vector. The UC column classifies the example (if applicable) into one of four use cases, following the ordering in Section 2.3. LOS and LOC indicate the (potentially overlapping) numbers of closure-related lines of specifications (in Prusti's actual syntax) and code, respectively. LOG counts the lines of glue code required by Prusti to deal with unsupported Rust features, e.g. library functions. VT gives the average verification time in seconds over 10 runs. `fold_list_rev` has been manually encoded into Viper and is not comparable to the other examples.

Example	CS			HO			NC	UC	LOS	LOC	LOG	VT
	A	M	HI	SE	CD	GA						
`counter` `counter_err`	✓	✓	✓	✓			≥		8	10 9	1	5.570 5.120
`delegation`	✓	✓		✓	✓		=		9	20	1	5.967
`blameassgn` `blameassgn_err`				✓			=		11	12 14	1	4.881 5.344
`option` `option_map_err`	✓	✓		✓	✓		=	1	103 18	106 21	19	8.428 8.297
`map_vec`	✓	✓	✓	✓	✓		≥	1	34	29	36	12.662
`result_uoe`				✓	✓		=	2	16	14	29	6.768
`repeat_with_n`	✓	✓	✓	✓	✓		≥	2	19	21	36	12.740
`fold_list_rev` (Viper)	✓	✓		✓		✓	≥	3				4.193
`any` `any_err`				✓	✓		≥	4	38	33	36	10.770 10.693
`all`				✓	✓		≥	4	39	33	36	10.218

Our Prusti extension did not noticeably increase verification times compared to similarly sized examples without closures (cf. Table 1 on page 22 of Astrauskas et al. [2019]). This demonstrates that our methodology and encoding strategy are feasible for use in a practical program verifier. Moreover, the verification times for erroneous examples are very similar to those for their correct counterparts. The performance on erroneous examples is important for the practical use of a verifier, when it is run repeatedly on incorrect examples until all bugs are fixed and all necessary annotations provided. Our case studies required all specification constructs of our methodology (cf. Section 3) for their successful verification.

Examples in Detail. The example `counter` is inspired by Kassios and Müller [2010]; it uses a closure to increment a captured variable every time the closure is called. The closure is passed to a higher-order function that, in principle, could invoke it an arbitrary number of times, thereby demonstrating the need for history invariants. The specification overhead in the original paper is comparable to what is required in our implementation.

By contrast, verifying the `delegation` example, also taken from Kassios and Müller [2010], relies on complex framing conditions, which our approach can automatically derive from Rust's type system; the specification overhead is thus significantly smaller than in the original paper.

Findler and Felleisen [2002] have inspired the `blameassgn` example, which we have fleshed out according to their description since they do not provide explicit code. Our implementation is capable of assigning the "blame" for a higher-order contract violation correctly and *statically,* whereas they perform dynamic checking with about the same amount of specification overhead.

For the `option` example, we have specified and verified all higher-order functions provided by the Rust standard library's `Option` type (monomorphised to `Option<i32>` for simplicity), together with several concrete use cases. Similarly, `result_uoe` models the `unwrap_or_else` higher-order function of Rust's `Result` type, which is used for error handling and reporting.

`map_vec` and `repeat_with_n` showcase common higher-order functions of Rust's `Iterator` type, although for reasons unrelated to closures and higher-order functions, we have chosen to work directly on vectors instead of the much more complex `Iterator` trait. `map_vec` demonstrates a pointwise side-effectful `map` operation on a `Vec<i32>`, and `repeat_with_n` performs the equivalent of an `iter::repeat_with()` call, followed directly by a `collect::<Vec<i32>>()` call to produce a vector. `any` and `all` similarly model the corresponding functions in the `Iterator` standard library trait, but again work directly on vectors for simplicity.

Finally, the `fold_list_rev` example concerns the list reversal algorithm that was analysed as a case study by Svendsen et al. [2010], without automation and in a higher-order specification logic. We were unable to encode this example directly for our implementation, due to reasons orthogonal to the verification of closures: in particular, missing support for reasoning about mathematical sequence types in Prusti. To demonstrate that our *methodology* is in principle capable of handling such data structure transformations involving closures, we manually encoded the example in the Viper intermediate verification language, following the encoding strategy in Section 4.

5.3 Use of Rust Closures in Practice

We performed a preliminary analysis of the 200 most commonly used Rust crates, i.e. binaries and libraries, from the Rust community's public registry https://crates.io using the Qrates querying infrastructure [Astrauskas et al. 2020] to substantiate our claim that the use cases discussed in Section 2 represent idiomatic usages of closures in Rust. The data we collected shows that several different versions of `map()` (→ point-wise transformations, Section 2.3.1), some functions of the `*_or_else()` kind (→ lazy generators, Section 2.3.2), as well as the `filter()`, `any()`, `all()`, and `find()` functions of `Iterator` (→ closures as predicates, Section 2.3.4) together make up the majority of all higher-order function calls in the real-world Rust code that we analysed. This suggests that our use cases from Section 2 are indeed very common in practice.

Additionally, we investigated which concrete closure types are passed into higher-order functions. Rust has three main closure-related traits: `Fn`, `FnMut`, and `FnOnce`, with `Fn <: FnMut` and `FnMut <: FnOnce`. The `Fn` trait is implemented by all closures which neither modify nor move their captured state; `FnMut` closures may modify but not move, and `FnOnce` closures may modify and move their captured state (and can thus only safely be called once). Using these trait bounds, a higher-order function can declare the most general type of closure it supports, and due to the subtyping relations between the traits, a caller may pass in a more specific type; for instance, `Iterator::map()` from Rust's standard library supports `FnMut` closures, but a caller can pass in, say, `|x| x * 2`, an `Fn` closure because it does not have mutable captured state.

We observed that in places where `FnMut` closures are allowed, roughly 10% of call sites supply an `FnMut` closure in practice; i.e. a minority but nonetheless significant portion of closures passed to higher-order functions likely *do* have mutable captured state. This justifies the efforts put into our methodology to support verification of closures with mutable captured state.

6 RELATED WORK

This article builds on the master's thesis of Wolff [2020]. Closest to our work is an investigation of closure verification by Kassios and Müller [2010], which overlaps with the work of Nordio et al. [2010], where verification of C# *delegates* is discussed. While our encoding strategy is similar, the above works neither consider advanced specification features like call descriptions, nor leverage a type system for framing guarantees.

Svendsen et al. [2010] discuss verification of C# delegates, but they resort to higher-order separation logic, making their approach unsuited for automatic verifiers based on SMT solvers. Several other treatments of higher-order function verification also apply higher-order logic [Krishnaswami 2012; Régis-Gianas and Pottier 2008].

An extensive exploration of the theory of imperative higher-order function verification using an extended Hoare logic is given by Honda et al. [2005] and Yoshida et al. [2007]. While achieving strong theoretical results, their use of Hoare logic with many non-structural rules (i.e. rules that do not just manipulate assertions) does not lend itself to implementation in an automatic verifier.

The problem of higher-order function verification naturally arises in functional programming languages. Nanevski et al. [2008a] present *Hoare Type Theory* as a means for verifying a functional programming language (extended with some imperative concepts, such as mutable state). This technique is mainly based on the *Hoare monad*, an extension of the state monad with pre- and postconditions. Swamy et al. [2013] likewise use the monadic approach for closure verification: they employ the so-called *Dijkstra monad* (a monad of predicate transformers describing the weakest precondition of a computation in any context in which it may appear), together with a type inference algorithm to compute verification conditions for functional programs written in F*. Their technique extends to stateful JavaScript programs via a translation from JavaScript to F*. Similarly to our technique, they consider the evolution of closures across multiple states; our history invariants are comparable to a variant of the Dijkstra monad that is equipped with a reflexive and transitive two-state invariant. In contrast to our technique, however, which is directly encoded in first-order logic, Swamy et al. [2013] encode to F*'s higher-order logic, representing a challenge for automation. In particular, their approach requires:

(1) monadic F* for expressing non-trivial framing properties, whereas we leverage Rust's type system to get such properties essentially for free;
(2) higher-order logic for expressing complex closure and higher-order function specifications, e.g. that a closure was called with certain arguments in some intermediate state to produce a certain effect, whereas we have introduced novel specification constructs, like call descriptions, to enable reasoning about such scenarios in a first-order setting; and
(3) user-supplied predicate transformers to eliminate expensive higher-order quantifications, whereas our technique is fully automated.

O'Hearn and Reynolds [2000] and Charguéraud and Pottier [2008] also perform translations from imperative to functional programming languages in order to gain insights about the semantics of the original program. Our methodology works directly on an imperative programming language, eliminating the need for complex transformations.

Some of the existing literature about verification of effectful higher-order functions is based on proof assistants, usually Coq. Nanevski et al. [2008b] present an extension (called *Ynot*) of Coq itself to allow for writing and reasoning about side-effectful higher-order functions inside Coq. Meanwhile, Kanig and Filliâtre [2009] present a custom language and tooling for describing and verifying effectful higher-order programs. Their tool eventually generates proof obligations to be manually discharged in Coq, whereas our tool generates first-order proof obligations, to be automatically discharged by an SMT solver.

Findler and Felleisen [2002] explore contracts for higher-order functions in Scheme, but their approach is based on runtime checks, whereas we perform static verification. They focus on the question of *blame assignment*, claiming that for a higher-order function *g* receiving a function argument, "a contract checker cannot ensure that *g*'s argument meets its contract when *g* is called" [Findler and Felleisen 2002, page 49]. This problem is solved by our specification entailments (Section 3.3), which do ensure at the higher-order function's call site that its argument functions satisfy their respective specifications.

Soundarajan and Fridella [2004] use trace-based reasoning to talk about which calls a function makes during its execution, but writing trace-based specifications is often unintuitive and complex (as amply demonstrated by their case study [Soundarajan and Fridella 2004, pages 321–329]).

Our key problem K6 (see Section 2.3.4) relates to reasoning about pure functions that also have executable implementations, as discussed by Darvas and Leino [2007]. In the future, we hope to extend our solution to this problem, perhaps with a notion of pure closures of some sort. Pereira [2018] presents some techniques for reasoning about higher-order accumulation functions such as `fold` (→ key problem K5), using a strategy very similar to ours, also consisting, essentially, of ghost argument functions to represent invariants and to avoid higher-order quantifications.

Finally, Shaner et al.'s *model programs* [Shaner et al. 2007] for the Java Modeling Language (JML) bear some similarity to our call descriptions (Section 3.4). Compared to model programs, we strive for a cleaner separation between code and specification, bearing in mind that model programs can easily lead to code duplication between implementation and specification, and therefore specifications that model a higher-order function's behaviour *too* closely, violating the principle of information hiding.

7 CONCLUSION AND FUTURE WORK

"The Rust programming language is fundamentally about *empowerment*" [Klabnik and Nichols 2021]. In the same vein, our methodology has been designed to empower programmers to formally reason about closures, by providing a high degree of abstraction and automation, and allowing programmers to write specifications at the Rust level, without having to deal with details of the underlying verification logic. We have achieved this goal by complementing the strong guarantees provided by Rust's type system with novel and expressive specification primitives such as specification entailment assertions and call descriptions.

As future work, we plan to generalise our call descriptions to conveniently express the *order* of closure calls in relation to each other, which is relevant for some examples where side effects on the captured state affect the results of subsequent closure invocations. Moreover, as briefly alluded to in Section 2.3.5, we plan to extend the techniques presented in this paper to deal with concurrent Rust programs in which closures are used to spawn new threads.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback. We would also like to thank Jichun Wu for helping us collect data on the usage of Rust closures in practice. We thank Vytautas Astrauskas and Federico Poli for supporting us with the integration of our verification technique into the Prusti verifier. We acknowledge AWS for partially supporting our work.

REFERENCES

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (11 2020), 27 pages. https://doi.org/10.1145/3428204

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (Oct. 2019), 30 pages. https://doi.org/10.

1145/3360573

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects (LNCS, Vol. 4111)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer, Berlin, Heidelberg, 364–387. https://doi.org/10.1007/11804192_17

Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *ICFP*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/1411204.1411235

Ernie Cohen, Mark A. Hillebrand, Stephan Tobies, Michał Moskal, and Wolfram Schulte. 2015. Verifying C Programs: A VCC Tutorial. Retrieved 2021-04-11 from https://bit.ly/32BkCWN Working draft, version 0.2.

Ádám Darvas and K. Rustan M. Leino. 2007. Practical Reasoning About Invocations and Implementations of Pure Methods. In *Fundamental Approaches to Software Engineering (LNCS, Vol. 4422)*, Matthew B. Dwyer and Antónia Lopes (Eds.). Springer, Berlin, Heidelberg, 336–351. https://doi.org/10.1007/978-3-540-71289-3_26

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.), Vol. 4603. Springer, 183–198. https://doi.org/10.1007/978-3-540-73595-3_13

Krishna K. Dhara and Gary T. Leavens. 1996. Forcing behavioral subtyping through specification inheritance. In *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 258–267. https://doi.org/10.1109/ICSE.1996.493421

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ICFP*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

David Harel, Dexter Kozen, and Jerzy Tiuryn. 2002. Dynamic Logic. In *Handbook of Philosophical Logic*, Dov M. Gabbay and Franz Guenthner (Eds.). Handbook of Philosophical Logic, Vol. 4. Springer Netherlands, Dordrecht, 99–217. https://doi.org/10.1007/978-94-017-0456-4_2

Kohei Honda, Nobuko Yoshida, and Martin Berger. 2005. An Observationally Complete Program Logic for Imperative Higher-Order Frame Rules. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS '05)*. IEEE Computer Society, Los Alamitos, CA, USA, 270–279. https://doi.org/10.1109/LICS.2005.5

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM 2011 (LNCS, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

Johannes Kanig and Jean-Christophe Filliâtre. 2009. Who: a verifier for effectful higher-order programs. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML (ML '09)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/1596627.1596634

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, Berlin, Heidelberg, 268–283. https://doi.org/10.1007/11813040_19

Ioannis T. Kassios and Peter Müller. 2010. *Specification and verification of closures.* Technical Report. ETH Zürich. https://doi.org/10.3929/ETHZ-A-006843251

Steve Klabnik and Carol Nichols. 2021. The Rust Programming Language. Retrieved 2021-04-13 from https://doc.rust-lang.org/book/

Neelakantan R. Krishnaswami. 2012. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic.* Ph.D. Dissertation. Carnegie Mellon University. https://doi.org/10.1184/R1/6724235.v1

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1 Jan. 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

Gary T. Leavens and David A. Naumann. 2015. Behavioral Subtyping, Specification Inheritance, and Modular Reasoning. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 13 (Aug. 2015), 88 pages. https://doi.org/10.1145/2766446

Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M Zimmerman. 2008. JML reference manual.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LNCS, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

Nicholas D. Matsakis. 2013. The Case of the Recurring Closure. Retrieved 2021-08-13 from http://smallcultfollowing.com/babysteps/blog/2013/04/30/the-case-of-the-recurring-closure

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the use of lambda expressions in Java. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 85 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133909

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (LNCS, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, Berlin, Heidelberg, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008a. Hoare type theory, polymorphism and separation. *J. Funct. Prog.* 18, 5–6 (Sept. 2008), 865–911. https://doi.org/10.1017/S0956796808006953

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008b. Ynot: dependent types for imperative programs. In *ICFP*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1411204.1411237

Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. 2010. Reasoning about Function Objects. In *Objects, Models, Components, Patterns (LNCS, Vol. 6141)*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 79–96. https://doi.org/10.1007/978-3-642-13953-6_5

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic (LNCS, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, Berlin, Heidelberg, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Peter W. O'Hearn and John C. Reynolds. 2000. From Algol to polymorphic linear lambda-calculus. *J. ACM* 47, 1 (Jan. 2000), 167–223. https://doi.org/10.1145/331605.331611

Mário José Parreira Pereira. 2018. *Tools and Techniques for the Verification of Modular Stateful Code.* Ph.D. Dissertation. Université Paris Saclay. Retrieved 2021-04-15 from https://tel.archives-ouvertes.fr/tel-01980343/document

Yann Régis-Gianas and François Pottier. 2008. A Hoare Logic for Call-by-Value Functional Programs. In *Mathematics of Program Construction (LNCS, Vol. 5133)*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer, Berlin, Heidelberg, 305–335. https://doi.org/10.1007/978-3-540-70594-9_17

Steve M. Shaner, Gary T. Leavens, and David A. Naumann. 2007. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA*. Association for Computing Machinery, New York, NY, USA, 351–368. https://doi.org/10.1145/1297027.1297053

Jan Smans, Bart Jacobs, and Frank Piessens. 2010. Heap-Dependent Expressions in Separation Logic. In *Formal Techniques for Distributed Systems (LNCS, Vol. 6117)*, John Hatcliff and Elena Zucca (Eds.). Springer, Berlin, Heidelberg, 170–185. https://doi.org/10.1007/978-3-642-13464-7_14

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.* 34, 1, Article 2 (May 2012), 58 pages. https://doi.org/10.1145/2160910.2160911

Neelam Soundarajan and Stephen Fridella. 2004. Incremental Reasoning for Object Oriented Systems. In *From Object-Orientation to Formal Methods (LNCS, Vol. 2635)*, Olaf Owe, Stein Krogdahl, and Tom Lyche (Eds.). Springer, Berlin, Heidelberg, 302–333. https://doi.org/10.1007/978-3-540-39993-3_15

Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2010. Verifying Generics and Delegates. In *ECOOP 2010 – Object-Oriented Programming (LNCS, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 175–199. https://doi.org/10.1007/978-3-642-14107-2_9

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. *SIGPLAN Not.* 48, 6 (June 2013), 387–398. https://doi.org/10.1145/2499370.2491978

Fabian Wolff. 2020. *Verification of Closures in Rust Programs.* Master's thesis. ETH Zürich. https://doi.org/10.3929/ethz-b-000444764

Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021a. *Modular Specification and Verification of Closures in Rust (artefact).* https://doi.org/10.5281/zenodo.5482557

Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021b. Prusti extension for closure verification. https://github.com/Aurel300/prusti-dev/tree/closure-oopsla-submission-2.

Nobuko Yoshida, Kohei Honda, and Martin Berger. 2007. Logical Reasoning for Higher-Order Functions with Local State. In *Foundations of Software Science and Computational Structures (LNCS, Vol. 4423)*, Helmut Seidl (Ed.). Springer, Berlin, Heidelberg, 361–377. https://doi.org/10.1007/978-3-540-71389-0_26