Leveraging rust types for modular specification and verification

Leveraging Rust Types for Modular Specification and Verification

VYTAUTAS ASTRAUSKAS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
FEDERICO POLI, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, ETH Zurich, Switzerland

Rust's type system ensures memory safety: well-typed Rust programs are guaranteed to not exhibit problems such as dangling pointers, data races, and unexpected side effects through aliased references. Ensuring correctness properties beyond memory safety, for instance, the guaranteed absence of assertion failures or more-general functional correctness, requires static program verification. For traditional system programming languages, formal verification is notoriously difficult and requires complex specifications and logics to reason about pointers, aliasing, and side effects on mutable state. This complexity is a major obstacle to the more-widespread verification of system software.

In this paper, we present a novel verification technique that leverages Rust's type system to greatly simplify the specification and verification of system software written in Rust. We analyse information from the Rust compiler and synthesise a corresponding *core proof* for the program in a flavour of separation logic tailored to automation. To verify correctness properties beyond memory safety, users can annotate Rust programs with specifications at the abstraction level of Rust expressions; our technique weaves them into the core proof to verify modularly whether these specifications hold. Crucially, our proofs are constructed and checked automatically without exposing the underlying formal logic, allowing users to work exclusively at the level of abstraction of the programming language. As such, our work enables a new kind of verification tool, with the potential to impact a wide audience and allow the Rust community to benefit from state-of-the-art verification techniques. We have implemented our techniques for a subset of Rust; our evaluation on several thousand functions from widely-used Rust crates demonstrates its effectiveness.

1 INTRODUCTION

Producing reliable system software is challenging. Pointer manipulation, mutable heap data and concurrency are typically employed to achieve high performance, but cause subtle bugs that are notoriously difficult to uncover and reproduce.

The Rust programming language addresses this problem by preventing some errors statically through its type system, which associates an *exclusive capability* [Boyland et al. 2001] with each mutable memory location. At any time, each exclusive capability is held by at most one executing function: only that code may access the memory location. When aliasing is desired, these exclusive capabilities can be exchanged for *shared capabilities*, with which many references can read a location, but none can modify it. Rust's type system enforces this discipline, ensuring that well-typed Rust programs are guaranteed to not exhibit data races, have dangling pointers or unexpected side effects through aliased references.

Going beyond memory safety, to guarantee absence of assertion failures or to prove functional correctness, requires static program verification. Despite recent successes [Bhargavan et al. 2017; Hawblitzel et al. 2015, 2014; Klein et al. 2009], formal verification of system software is notoriously difficult. Reasoning about pointers, aliasing, mutable state, and concurrency requires complex program logics, often based on separation logic [O'Hearn 2004; Reynolds 2002], dynamic frames [Kassios 2011; Leino 2010], or object ownership [Cohen et al. 2009; Leino and Müller 2004]. The expressive power of such logics comes at a price: they describe program behaviours via a rich language of custom assertions (e.g. the points-to predicates, separating conjunction, and magic wands of separation logic). Users are forced to understand these logics to write specifications and

direct the construction of a suitable proof. Furthermore, these logics typically require a substantial initial specification effort, even to prove simple properties such as crash-freedom or absence of overflow. Consequently, the application of these logics remains the domain of expert researchers, forming a major obstacle to the more-widespread verification of system software.

In this paper, we present a novel verification technique that leverages Rust's type system to greatly simplify the specification and verification of Rust programs. Our key insight is to combine the rich capability information implicit in Rust's type system with user-provided assertions that express functional behaviour to automatically construct a proof in an expressive program logic. We analyse information from the Rust compiler and synthesise a *core proof* of memory safety of the program in a flavour of separation logic that facilitates the integration of functional correctness properties. Crucially, our proofs are constructed and checked *automatically*; users of our technique never work with the underlying formal logic. They can add specifications at the abstraction level of Rust expressions; our technique interweaves these specifications automatically into the core proof to verify them modularly. Consequently, our technique shields users completely from the complexity of the underlying logic; assertions and error messages are expressed at the level of Rust expressions, which makes our technique accessible to programmers.

Contributions. The main contributions of our work are:

(1) We define a specification language for expressing functional properties of Rust programs, suitable for modular verification. Our language is based on Rust expressions, and does not expose the complexity of the underlying verification logic.
(2) We propose *pledges*: a novel specification construct that enables modular specification and verification of Rust functions that yield borrowed references.
(3) We define a verification technique that encodes both capability information and user-provided assertions into the implicit dynamic frames logic [Smans et al. 2012], a close relative of separation logic [Parkinson and Summers 2012].
(4) We automate our verification technique by constructing a translation from the Rust program and specifications into the Viper intermediate verification language [Müller et al. 2016]. Our translation generates correct specifications and, crucially, synthesises all necessary auxiliary annotations needed for proof checking to be completely automatic.
(5) We provide an implementation of our technique as a plugin for the Rust compiler. We used our implementation to automatically construct core proofs for several thousand unannotated Rust functions, and to verify a range of stronger properties (via our specification language) for selected Rust implementations. We will submit our tool as an artifact.

Unsafe Code and Rustbelt. Rust's type system enforces strong rules, but provides an escape hatch for when these are too restrictive: code blocks and functions can be declared *unsafe*, weakening the compiler's checks, and correspondingly risking the guarantees they provide. Unsafe code should be encapsulated by libraries such that client code cannot observe its usage [Rust contributors 2019b]. The ongoing Rustbelt project [Jung et al. 2017] is aimed at defining formal semantic foundations for making this requirement precise and verifiable. Our work has fundamentally different (and complementary) aims and technical contributions. We do not address unsafe code in this paper, but present a verification technique enabling user-specifications at a high level of abstraction and automatic proofs; Rustbelt verification entails directly using an advanced separation logic based on Iris [Jung et al. 2015], for which proofs are interactive (in Coq [Coq Team 2014]), and constructed by experts (see also Sec. 8).

Outline. The rest of this paper is organised as follows. We illustrate our approach on an example in Sec. 2. Sec. 3 presents our specification and verification technique for Rust without references;

```
1  struct Point {
2    x: i32, y: i32
3  }
4
5  #[ensures="p.x == old(p.x) + s"]
6  #[ensures="p.y == old(p.y)"]
7  fn shift_x(p: &mut Point, s: i32) {
8    p.x = p.x + s
9  }
```

```
10 fn compress(
11   mut segm: (Box<Point>, Box<Point>)
12 ) -> (Box<Point>, Box<Point>)
13 {
14   let diff = (*segm.0).x - (*segm.1).x;
15   shift_x(&mut segm.1, diff + 1);
16   assert!((*segm.0).x < (*segm.1).x);
17   segm
18 }
```

Fig. 1. Points in Rust. Proving that the assertion holds requires properties guaranteed by the Rust ownership system as well as a user-provided specification for function `shift_x`.

Secs. 4 and 5 extend our technique to handle mutable and shared references, respectively. Sec. 6 introduces pledges, used to attach functional specifications to mutable borrows. We describe and evaluate our implementation in Sec. 7, discuss related work in Sec. 8, and conclude in Sec. 9. Appendix A includes a detailed illustration of our encoding on a simple example.

2 MOTIVATING EXAMPLE

In this section, we illustrate the basics of our approach from a programmer's perspective. Details are explained in subsequent sections.

Example. Fig. 1 shows a simple Rust program, which declares a struct `Point` with two integer fields, and two functions. The function `shift_x`, shifts the x-coordinate of a given `Point` instance. Rust types express capabilities to access memory. Here, the type `&mut Point` expresses that `p` is a mutable reference, also called a *mutable borrow*. When the function is called, the capabilities to access the fields of the passed `Point` instance are *temporarily* transferred from the caller to the callee function, and back when the function terminates. Since the borrow is mutable, `shift_x` is allowed to modify the instance, here, by assigning to its `x` field.

Function `compress` takes the capabilities for a mutable pair of *boxed* points. A value of type `Box<T>` represents a pointer (with capabilities) to a value of type `T`; this indirection allows the `Points` to be passed by-reference (instead of by copying them). The selectors `.0` and `.1` select elements of the parameter pair.

The `assert!` statement performs a runtime check that its parameter evaluates to true. Evaluating `*(segm.1).x` here is allowed although `segm.1` was borrowed on the previous line (`&mut segm.1` creates a mutable reference to `segm.1`), as the compiler infers that the borrow is no longer used after the call. Therefore, the borrow *expires* after the call, restoring capabilities to the borrowed-from `segm.1`.

Correctness Arguments. Consider what is needed to prove that the `assert!` statement can *never* fail at runtime. Showing this property for *all* calls to `compress` requires the following properties, in which p_0 and p_1 denote the `Point` instances passed into function `compress` as `segm.0` and `segm.1`:

(1) The call to `shift_x` increases the value of p_1`.x` by the value of `diff + 1`.
(2) The call does not modify p_0`.x`. Therefore, right after the call, we have p_0`.x < `p_1`.x`.
(3) The call to `shift_x` does not modify the tuple `segm`, that is, we still have p_0`=segm.0` and p_1`=segm.1` and, therefore, `(*segm.0).x < (*segm.1).x`.
(4) The code is data race free and, thus, the values of all memory locations are stable throughout the execution.

Except for property 1, all of these properties are guaranteed by Rust's type system. In particular, `segm.0` and `segm.1` are guaranteed to reference *different* `Point` instances p_0 and p_1; since only the capabilities for p_1 are transferred to function `shift_x`, all fields of p_0 are guaranteed to be left unchanged by the call (property 2, and analogously for property 3). Preserving information about mutable state, so-called *framing*, is one of the main difficulties of modular verification [Kassios 2011; Leino and Nelson 2002; Reynolds 2002]; by leveraging information from Rust's type system, our technique solves the frame problem without imposing substantial overhead on programmers.

Property 4 is a consequence of the fact that Rust's type system requires an exclusive capability in order to mutate a memory location. It allows one to verify the assertion without reasoning about thread interleavings [Jones 1983; Owicki and Gries 1976] or explicit proofs of race freedom [O'Hearn 2004], which would increase the specification effort for programmers.

Property 1 follows from the functional behaviour of function `shift_x`, which is expressed as a user-provided postcondition, written as a Rust annotation. Our specification language is based on Rust boolean expressions, extended with few (but powerful) additional constructs; here, the `old` construct [Leavens et al. 2011] is used to refer to the pre-state value of a mutable memory location, which allows one to express relational properties between the pre- and post-state of a call. The second postcondition of `shift_x` is not needed to verify the assertion, but would likely be required by other client code. Since `shift_x` takes a mutable borrow to the `Point` instance `p`, the type system allows it to modify any fields of `p`. The second postcondition tightens framing by guaranteeing that `p.y` will remain unchanged. Note that our specifications are as simple as traditional contracts [Meyer 1992], but enable the sound verification of concurrent, heap-manipulating programs.

The assertion could in principle be proved without the postconditions, by inlining the implementation of `shift_x`. However, verifying a call against a specification, instead of an implementation, makes verification *modular*, which is important for scalability, to provide guarantees for library code, and to reduce and localise the re-verification effort when parts of a codebase are changed.

Verification. Our example illustrates that correctness proofs for Rust programs need to combine information about capabilities for memory locations (aliasing, side effects, framing, data race freedom) with information about their values. While the former is provided by the type system, the latter must be supplied as assertions (the *inference* of value information is possible, but orthogonal). To formally integrate both sources of information, our technique encodes capabilities and user-provided functional properties into a program logic that is sufficiently expressive to capture both and reason about their interactions.

Our verification technique encodes the capability information and statement semantics of Rust programs into a formal logic, resulting in a *core proof* that captures information about aliasing and side effects that is essential for verification, in particular, for framing. It is crucial that this encoding, as well as the *checking* of our core proofs, is completely automatic; any required user interaction would expose the complexity of the underlying program logic to the programmer and, thereby, break the abstraction that our work aims to provide. For program logics (such as separation logics) expressive enough to model Rust's type capabilities, this degree of automation is beyond the state of the art.

Our core proofs provide the technical foundations for verifying stronger properties, such as the correctness of user-provided assertions, and absence of arithmetic overflows and various kinds of exceptions (called *panics* in Rust), including **`assert!`** failures. Constructing the core proof is challenging, especially handling complex forms of (re)borrowing and synthesising auxiliary annotations to automate the proof search, as we will explain later.

A major virtue of our approach is that it lowers the barrier to applying verification. The construction of the core proof from a well-typed Rust program is fully automatic, such that programmers

```
1  #[ensures="old((*p).x + s) == (*result).x"]
2  #[ensures="old((*p).y) == (*result).y"]
3  fn shift_x(p: Box<Point>, s:i32) -> Box<Point> {
4    box Point { x: (*p).x + s, y: (*p).y }
5  }
6
7  fn compress(mut segm: (Box<Point>, Box<Point>)) -> (Box<Point>, Box<Point>) {
8    let mut end = segm.1; // move assignment
9    // segm.1 is now inaccessible
10   let diff = (*segm.0).x - (*end).x;
11   end = shift_x(end, diff + 1);
12   segm.1 = end;
13   // end is now inaccessible
14   assert!((*segm.0).x < (*segm.1).x);
15   segm
16 }
```

Fig. 2. A variation of the example from Fig. 1 that uses move assignments instead of borrowing. The move assignment in line 8 removes the capability for `segm.1` until it is restored in line 12, which prevents accesses in between. In particular, omitting line 12 would cause a compiler error, since it would not be possible to assemble the full capabilities required by the return type.

can immediately focus on verifying the main properties of interest, such as the validity of a given assertion. They can control the required effort by writing simpler or more comprehensive specifications; if the (optional) checks for built-in properties such as overflow are not enabled, the minimal specification is none at all. This is in stark contrast to most existing verification techniques, which require a substantial initial effort to set up predicates, invariants, or ghost state and to verify memory safety, before programmers can turn to the properties they care about most.

3 RUST'S CAPABILITIES FOR VERIFICATION

In this section, we explain our specification and verification technique for Rust code without borrowing, which we defer until Secs. 4 and 5. We present the capability information that is needed to construct a core proof, explain how we encode this proof to Viper, and then show how to incorporate user-provided assertions.

We present our work for a small but technically-challenging subset of safe Rust. It includes: **`bool`**, integer, **`char`**, **`struct`** and **`enum`** types, move and copy assignments, heap-allocated data (including **`Box`** types), shared and mutable borrows (including reborrowing), traits, generics, loops, and function calls, including common use-cases of lifetime parameters to functions. Commonly-used Rust features which fall outside of our subset include: lifetime parameters to **`struct`** types, as well as unsafe code; these are left for future work.

3.1 Ownership and Capabilities

The Rust type system enforces a strict discipline governing not only which values can be stored in which locations, but also which *places* (Rust's terminology for expressions denoting memory locations [Rust community 2018b]) can be used to access those values at each program point.

Ownership. In Rust, every value stored in memory has a unique *owner*, which is a variable (variables always include function parameters) in a currently-active function execution. Ownership is transitive: the owner of a struct value is also the owner of its fields. The scope of a value's owner implicitly determines the deallocation time of the owned memory. Rust's type rules guarantee that

by the time the owner goes out of scope (or if the owning variable is reassigned), no place will have the capability to access the underlying memory, preventing dangling pointers. Rust types typically convey ownership of the corresponding memory; for instance, **Box**<T> is the type of an *owning* pointer.

Fig. 2 shows a variation of the example from Fig. 1 without borrowing (which we will explain shortly). The assignment in line 8 is a *move assignment*, which transfers ownership from `segm.1` to `end`, making `segm.1` unusable. Similarly, the call to `shift_x` transfers ownership from `end` to parameter `p`. `end` becomes usable again (and owns its contents) on its reassignment when the function terminates, and analogously for `segm.1` in line 12. The subsequent assertion holds for reasons similar to those outlined in the previous section for Fig. 1. In particular, ownership guarantees that the two points are distinct objects, and provides framing for the call to `shift_x`.

Capabilities. Owning a memory location does not necessarily provide the right to access it. For instance, function `compress` in Fig. 1 owns both points throughout its execution, but the right to access the point in `segm.1` is *temporarily* transferred to `shift_x` using a borrow[1]. Borrowing affects who may access a location, but not who owns it. To distinguish these concepts, we use the term *place capability* (or *capability* for short) to denote the right to access the value stored in a place.

Precise knowledge of the capabilities at any given program point is crucial for verification, especially framing. For instance, function `compress` in Fig. 1 may frame the value of `(*segm.0).x` around the call to `shift_x` because it retains the corresponding capability, whereas `(*segm.1).x` may change because the capability is transferred for the call. Note that Rust source types do not provide complete capability information: for instance, throughout a function body, struct-typed variables retain the same Rust type, but capabilities to their fields vary as they are borrowed or moved. To make this information explicit, we defined an algorithm to compute precise summaries of the capabilities held at each program point, which we call place capability sets.

In the following, we define the type of *results* our algorithm computes, but omit the algorithm itself for brevity. At verification boundaries such as function pre- and post-states, we extract these results directly from the Rust compiler: we use the declared types of all definitely-assigned variables in scope at these program points to compute a suitable summary of the capabilities held. However, to elaborate this to an automatable formal proof we need explicit information about how these capabilities evolve at each *intermediate* program point. This is information which, in principle, is internally computed by the Rust type checker, but using representations which are not exposed; our algorithm therefore recovers these intermediate steps to produce a detailed account of the capabilities at every program point.

Definition 3.1 (Place Capability Sets). *Places*, ranged over by p, are expressions defined by the following grammar: $p ::= x \mid p.f \mid (*p)$. For a place p of the form $p'.f$ and $(*p')$, place p' is called a *sub-place* of p; this notion is extended transitively. A *place capability set* (PCS) is a finite set of places.

The initial PCS for a function contains exactly the capabilities for its parameters: for instance, {`segm`} for function `compress` in Fig. 2. Every subsequent statement may require certain capabilities to be in the PCS and then transform the PCS. For instance, the move assignment on line 8 requires the PCS before the assignment to contain `segm.1`, and transforms the PCS from {`segm.0`,`segm.1`} to {`segm.0`,`end`}, reflecting the move of capabilities. These PCS transformations are defined for each primitive statement; we provide details in App. B.

The evolution of capabilities during Rust type checking can require additional operations on top of the requirements for individual Rust statements. For example, the PCS before the assignment on

[1]In this and the next section, we will discuss only exclusive capabilities; we extend our work to shared capabilities in Sec. 5.

line 8 is obtained from the initial PCS of the function by exploiting transitivity of capabilities. Since the capabilities of a place also imply capabilities for all its sub-places, the type checker can *unpack* the capability for `segm` into `{segm.0,segm.1}`. Unpacking is one of several PCS operations the type checker may perform to manipulate PCSs:

Definition 3.2 (PCS Operations). A *PCS operation* is a *remove*, *unpack*, or *pack* of a capability in a PCS. Remove is defined as the corresponding set operation.

Let p be a place of struct type, and let $f_1, \ldots, f_n$ be the fields of the struct. For a PCS S such that $p \in S$, the *unpacking* of p in S is the PCS $(S \setminus \{p\}) \cup \{p.f_1, \ldots, p.f_n\}$. If p is instead of box type, the unpacking is $(S \setminus \{p\}) \cup \{(*p)\}$.

The *packing* of p in S is the inverse operation. It is defined only when the $p.f_i$ (or $*p$) are in S.

The Rust type checker implicitly employs these operations between statements to show that the capabilities required by the next statement are present. Unpack is used to enable operations on individual fields of structs (e.g. the move on line 8), and pack is used when the entire struct is passed as argument or result to check that capabilities for all sub-places can be reassembled, e.g. at the end of `compress` in Fig. 2. All three operations can be used at join points in the control flow if the joined paths provide different capabilities: remove is needed to drop the capabilities available in one path but not the other (e.g. due to moving out a struct field in a branch), while pack and unpack are needed to unify the PCSs of the joining paths.

By combining type information extracted from the compiler and our own analysis, we infer automatically, for each statement, the PCS before the statement, a sequence of *necessary* PCS operations to be applied before the statement, and the PCS after the statement, describing the actual flow of capabilities implied by Rust's type rules. This information is vital for the construction of the core proof, as we explain in the next subsection. Since it provides a detailed account of *why* a Rust program type-checks, we believe that it could also be repurposed as the basis of other analysis, verification and visualisation tools.

3.2 Constructing the Core Proof

We verify Rust programs by encoding the program, capability information, and user-provided specifications into the intermediate verification language Viper [Müller et al. 2016], and using Viper's existing verification tools. Viper provides a simple heap-based imperative language, along with a number of reasoning primitives for expressing verification problems; each Viper method is equipped with a precondition and a postcondition; Viper loops are equipped with loop invariants. For each function in the Rust program, we generate a corresponding method in our Viper program, such that successful verification of the Viper method implies correctness of the Rust original.

Viper Resources. Viper's heap is object-based: heap locations are identified by a pair of a **`Ref`**-typed value and a field name. Viper's type system is simple: the built-in **`Ref`** type is the only type for objects in the Viper heap, and all fields declared in a Viper program are in principle available in all objects. Akin to separation logic, Viper enforces that a field location can be accessed only when *permission* is held to do so. Conversely, so long as the permission to a field location is held, Viper assumes that its value cannot change, which provides framing. Viper field permissions are tracked in the program state as *affine resources*; they can be explicitly added or removed from a program state, or implicitly dropped if not required.

Viper's logic is based on implicit dynamic frames [Smans et al. 2009], a close relative of separation logic [Parkinson and Summers 2012], but with the important facility to incorporate heap-dependent expressions in logical assertions (including calls to side-effect-free functions) [Smans et al. 2010]. Assertions called *accessibility predicates*, written **`acc`**`(e.f)` are used to denote the *exclusive field*

permission for the field `f` of the object denoted by `e`. Viper's conjunction `&&` acts *multiplicatively* (in the sense of linear logic [Girard 1987]); analogously to separating conjunction in separation logic, it requires the *sum* of the necessary resources in each conjunct. For example, the assertion `acc(x.f) && acc(y.f)` denotes *two* exclusive permissions, which implies that `x` and `y` cannot alias. In addition to accessibility predicates, our work makes crucial use of two other kinds of Viper resource assertions adopted from separation logic: predicates and magic wands, which will be explained later.

Modelling Memory. We model Rust's program states in Viper by mapping every Rust memory location to a corresponding Viper field location. We model any non-primitive type in Rust as a Viper object (of **`Ref`** type); each transitive element of the Rust type (struct fields, tuple elements, box contents, reference targets) is modelled as a field of the Viper object. Since Rust references make it possible to take the address at which any value is stored, we also model Rust primitive types with an additional indirection; any primitive type is modelled as a Viper object with a single field that contains the actual value.

As an example, the parameter `segm` in Fig. 2 is modelled as a Viper **`Ref`** with fields `elt0: Ref`, `elt1: Ref` for the two elements of the tuple; in turn, each of these values is a box, modelled as an object with a single field `val_ref: Ref`. Similarly to pairs, `Point` struct values are Viper objects with two fields, while their individual **`i32`** fields (also addressable in Rust) are modelled as objects with a single field `val_i32: Int`. Here, we use Viper's built-in **`Int`** type for unbounded integers; if overflow-checking is enabled, we encode bounds using additional assertions.

Modelling Rust Types. As explained above the core proof requires precise capability information, for instance, to enable sound framing. To provide this information, we model the capabilities represented by Rust types as resource assertions in Viper. Since place capabilities can have unboundedly many sub-places (struct types may recurse via e.g. box types), we cannot enumerate these explicitly. Instead, we translate each Rust type into an instance of a Viper *predicate*. Predicates are a standard means of defining parameterised, possibly recursive assertions [Parkinson and Bierman 2005]; predicate *instances* are tracked as affine resources in Viper.

We define a Viper predicate per Rust type in the source program; each predicate is parameterised by a single **`Ref`**-typed parameter (the Viper object representing the Rust place): for primitive types, the body contains an accessibility predicate for the single field storing the value, while for structures and tuples, it consists of the conjunction of accessibility predicates for each field, as well as a predicate instance for the translation of the field's type. As a simple example, we translate a place capability of type **`i32`** using the following `i32` predicate (the bounds properties within the predicate body are omitted if overflow-checking is disabled):

```
field val_i32 : Int

predicate i32(self: Ref) {
  acc(self.val_i32) && i32MIN <= self.val_i32 <= i32MAX
}
```

For polymorphic types such as **`Box`**`<T>` where the type parameter `T` is known we monomorphise, generating a specialised predicate e.g. for **`Box`**`<Point>`, where `Point` is the predicate generated for the Rust `Point` type:

```
predicate BoxPoint(self: Ref) {
  acc(self.val_ref) && Point(self.val_ref)
}
```

When the type parameter of a polymorphic type is not known, e.g. when encoding **Box**<T> in a generic function under a parameter T, we encode the type parameter as an abstract predicate, whose body is unspecified. For example, the encoding of **Box**<T> becomes:

```
predicate T(self: Ref);

predicate BoxT(self: Ref) {
  acc(self.val_ref) && T(self.val_ref)
}
```

The predicate instance `T(self.val_ref)` represents exclusive capability to the boxed value; by making the predicate abstract, it conveys no further information about this value.

For enumeration types (a form of tagged unions) we model the discriminant (tag) of the enumeration as an integer field with bounded values. This discriminant is then used on the left-hand-side of implications to guard which accessibility predicates for the variant's fields are actually included in the predicate. An example is provided for the **Option**<T> type in App. A.

Modelling Place Capabilities. As explained in the previous subsection, Rust types prescribe the available capabilities at function boundaries, but the PCS may change throughout the function. Using the predicates defined above, we can directly translate a place capability set into a corresponding Viper assertion: each element of the PCS gives rise to an instance of the Viper predicate corresponding to its type. We call the Viper assertion consisting of the conjunction of these predicate instances the *Viper embedding of the PCS*.

This embedding allows us to map each Rust function to a corresponding Viper method, along with corresponding preconditions and postconditions at the Viper level. The precondition is the Viper embedding of the PCS representing all input parameters to the Rust function; the postcondition is the Viper embedding of the PCS representing the output parameters. We use the precondition to prescribe the initial state for verifying the Viper method, while the postcondition is checked at the end of the method body. Analogously, the Viper embedding of the PCS at each loop head provides a loop invariant that lets Viper verify loops for our core proof automatically.

As an example, when we generate a Viper method for the Rust function `compress` in Fig. 2, the precondition will be `PairBoxPoint(segm)`, where `PairBoxPoint` is a Viper predicate whose body contains permission to the pair's fields and a `BoxPoint` predicate instance for each. Analogously, the postcondition will be `PairBoxPoint(result)`, where **result** refers to the method's return value.

Many program verifiers, including Viper, prevent indefinite unrolling of recursive predicates by treating predicates *isorecursively* [Abadi and Fiore 1996; Crary et al. 1999; Summers and Drossopoulou 2013]: exchanging a predicate instance for its body is not done automatically, but requires explicit operations in the program, called **fold** and **unfold** in Viper. These statements are needed exactly at those program points at which the Rust type checker performs the packing and unpacking PCS operations. By exploiting the PCS information we summarise for the Rust program, we are therefore able to instrument our generated Viper program with exactly the necessary additional annotations required for Viper to be able to reason about these predicates fully automatically, without any user interaction. Recall that full automation is essential to preserve the abstraction provided by our work and shield programmers from the complexity of the underlying logic employed in Viper.

Modelling Capability Transfer. In order to model Rust function calls, we need to correctly reflect the transfer of place capabilities. Having generated Viper methods, along with appropriate preconditions and postconditions, we want to model a call by *removing* the Viper resources in the

precondition, and subsequently *adding* the Viper resources in the postcondition. Viper provides `inhale` and `exhale` statements for such explicit manipulation of resources [Müller et al. 2016]. A statement **`exhale`** `A` has the effect of checking that the assertion `A` is true in the current state, and removing all resources it requires. Moreover, when permission to a memory location is removed, Viper removes any knowledge about the value stored in the location to reflect that the value could be changed by another function. Dually, **`inhale`** `A` adds the resources prescribed by `A`.

Using these Viper statements, we encode a Rust-level function call as an `exhale` of the precondition (reflecting that the corresponding capabilities become unavailable to the caller), followed by an `inhale` of the postcondition (reflecting those which are returned). Via the capability information extracted from the Rust compiler (in PCS form) and our Viper embedding of this information, this handling of Viper resources corresponds precisely to the transfer of capabilities in Rust. More details are illustrated in App. A.

3.3 Functional Specifications

The core proof we have constructed so far, by itself, does not go beyond the guarantees provided by the type system. However, it provides the foundation for verifying stronger properties such as functional correctness. In particular, it provides precise aliasing information and framing, which is essential for sound and modular reasoning about the Rust heap. Due to the design of our modelling of Rust types, and choice of underlying logic, extending our core proof to properties beyond memory safety is surprisingly simple.

We enable optional checking of generic properties such as absence of overflows and absence of panics (e.g. assertion failures) simply, by generating additional assertions in the Viper program. For example, to prove absence of assertion failures, it suffices to insert an **`assert false`** statement into the branch of the code that raises a panic when the runtime-check fails, to verify that this branch is unreachable.

User-provided assertions such as function pre and postconditions are translated and conjoined to the corresponding assertions of the core proof. This simple treatment is enabled by our choice of implicit dynamic frames for the underlying logic: unlike standard separation logics, implicit dynamic frames separates resource properties from value properties, as in **`acc`**`(x.f) && x.f > 0`. Similarly, predicate instances can be combined with applications of heap-dependent mathematical functions to constrain the resources in the predicate. We use this feature to allow user-provided specifications to include calls to side-effect-free Rust functions, similarly to JML's pure methods [Leavens et al. 2011], which is useful to express properties of unbounded data structures and to make use of abstractions already provided in the Rust program.

The technique presented so far supports the specification and verification of programs using only move and copy assignments. The treatment of borrowing is more intricate, both for the core proof and for functional specifications, as explained in the next sections.

4 MUTABLE BORROWS

One of the most important and intricate features of Rust's type system is *borrowing*: creation of references that temporarily take capabilities, but do not change ownership of the referenced value. In this section, we extend the construction of the core proof to *mutable* borrows; shared (immutable) borrows are covered in the next section. The specification and verification of functional properties in the presence of borrows will be discussed in Sec. 6.

```
1  // List of Points
2  struct Route {
3    current: Point,
4    rest: Option<Box<Route>>
5  }
6
7  #[pure]
8  #[ensures="result > 0"]
9  fn length(r: &Route) -> i32 {
10   1 + match r.rest {
11     Some(box ref q) => length(q),
12     None => 0
13   }
14 }
15
16 #[pure]
17 #[requires="0 <= n && n < length(r)"]
18 fn nth_x(r: &Route, n: i32) -> i32
19 {
20   if n == 0 { r.current.x } else {
21     match r.rest {
22       Some(box ref q) =>
23         nth_x(q, n-1),
24       None => unreachable!()
25     }
26   }
27 }
28
29 #[requires="0 <= n && n < length(r)"]
30 #[ensures="result.x ==
31   old(nth_x(r, n))"]
32 #[ensures="???"] // See Sec. 6
33 fn nth_point(r:&mut Route, n: i32) ->
34  &mut Point {
35   if n == 0 { &mut r.current } else {
36     match r.rest {
37       Some(box ref mut q) =>
38         nth_point(q, n-1),
39       None => unreachable!()
40     }
41   }
42 }
43
44 #[requires="0 <= n && n < length(r)"]
45 #[ensures="length(r) ==
46   old(length(r))"]
47 #[ensures="nth_x(r, n) ==
48   old(nth_x(r, n)) + s"]
49 #[ensures="forall i: i32 ::
50   (0<=i && i<length(r) && i != n) ==>
51     nth_x(r, i) ==
52     old(nth_x(r, i))"]
53 fn shift_nth_x(r: &mut Route,
54                n: i32, s:i32) {
55   let p = nth_point(r, n);
56   shift_x(p, s);
57 }
```

Fig. 3. An implementation of routes (sequences of points from Fig. 1), illustrating borrows. Function `shift_nth_x` borrows a route from its caller. This reference is reborrowed in the call to `nth_point`, which returns a reference to a point in the route. Both borrows expire after the call to `shift_x` on line 56. Functions annotated with [**pure**] are side-effect-free, which can be used in specifications. The missing ??? specification will be explained in Sec. 6.

4.1 Borrows and Lifetimes

Fig. 3 shows an example built upon the `Point` example from Fig. 1. Function `shift_nth_x` borrows a route from its caller; that is, the capability for the parameter `r` is transferred to the function. Each borrow has a *lifetime*, computed by the Rust compiler, representing an extent in the program execution for which the borrow needs to remain live; a lifetime always includes at least all program points where the borrow is used. Note that lifetimes are typically not explicit in the program text, but chosen implicitly by the compiler. At the end of a borrow's lifetime, the borrow is said to *expire*, and the capabilities associtated with it are restored to the borrowed-from place. Since `r` in `shift_nth_x` is a function argument, its lifetime spans the entire function body.

Reborrowing. It is possible to *reborrow* either the full place or a sub-place of an existing borrow. The call to `nth_point` at line 55 of Fig. 3 implicitly reborrows `r` and transfers its capability to

that function. More interestingly, function `nth_point` creates a borrow to a sub-place of the route, namely its n^{th} point, and returns it to its caller; this reborrow's lifetime persists beyond the call in which it is made. Reflecting this possibility, after the call to `nth_point`, `r` is blocked from being used until `p` expires, since `p`'s capabilities could (and indeed are, in this example) be for a part of the same memory that `r` had capabilities to access; if `r` were usable, this would violate exclusivity of these capabilities.

Reborrowing extends the lifetime of the borrowed-from reference: the original borrow cannot expire until all (transitive) reborrows are known to have expired. In our example, the lifetime of the reborrow created for the call to `nth_point` is extended until the further reborrow `p` expires after the call to `shift_x`[2].

Borrow Information. As we have explained in the previous section, constructing the core proof for a Rust program requires precise capability information, which we have so far represented via place capability sets and place capability operations at each program point (see Sec. 3.1). This information is insufficient for programs with borrows; in particular, it does not explain how, when borrows expire, the capabilities (and corresponding permissions in our Viper encoding) are restored to where they were borrowed from. For this we need precise information about which borrows are active for which lifetimes, and which reborrow each other.

We obtain information about the lifetimes selected by the Rust compiler from the latest borrow checker implementation [Rust contributors 2019a] and additional compiler analyses. In some cases, we also need to fill in missing information to explain this information; in general, the Rust compiler tracks *negative* information (sufficient to check whether an error should be raised), but does not always store explicit *positive* information witnessing why type-checking succeeds. We represent the extracted information as follows: we assign identities to each borrow operation in a function, as well as every move assignment of a reference (which we treat as a further reborrow), and every function call which returns a borrow. In terms of these identifiers, we record the set of identifiers of borrows which are alive before each statement. Moreover, we extract a *reborrow relation*: a binary relation on borrow identifiers, indicating which borrows *may* directly reborrow from which.

4.2 Encoding Borrows as Resource Assertions

In this subsection, we explain how we encode the capabilities associated with a mutable borrow as well as those for the remainder of the place from which it was borrowed. The next subsection discusses how the core proof manipulates these capabilities when borrows are created or expire.

The place capabilities associated with a mutable reference are encoded analogously to those for a box type (*cf.* Sec. 3.2). We define a Viper predicate to represent each reference type used in the program. For instance, predicate `RefMutPoint` for mutable references to points includes an instance of predicate `Point` for the referenced point, stored with an extra indirection through `val_ref`:

```
predicate RefMutPoint(self: Ref) {
  acc(self.val_ref) && Point(self.val_ref)
}
```

When a Rust function returns a borrow, this must always be a reborrow (possibly transitively) of a borrow passed as one of the function's parameters. Rust does not allow returning borrows to memory owned within the function (for instance, a local variable) because the returned borrow would become a dangling pointer when the function returns and the owned memory is deallocated. We refer to functions which take a mutable borrow as parameter and return a borrow as *lenders*.

[2]The exact rules depend on the version of Rust; in Rust 1.0, only entire explicit scopes (with a few exceptions) can be used as lifetimes. The recently introduced *non-lexical lifetimes* [Rust community 2017] are more fine-grained; we support both.

Intuitively, when a lender function such as `nth_point` creates and returns a borrow then it takes the capability for a parameter place (here, `r`) and splits it into two parts: the capability for the borrow *returned* by the function and any *remainder* of the original capabilities. Rust does not provide a way of representing such types with missing capabilities, nor are they used at function boundaries in Rust's type checking; instead, the borrowed-from place (including this remainder) is simply unusable until the returned borrow expires. However, to track information about the values stored in the borrowed-from data structure, we need to represent this remainder formally in our core proof and, thus, need a suitable formal model for these remainder capabilities.

Our key insight here is that the separation logic magic-wand connective [O'Hearn et al. 2001] lets us formally model the remainder capabilities resulting from a reborrow; it can express *partial* permissions to data structures, such as the `Route` with one `Point` missing. A magic wand assertion $A \mathrel{-\!\!*} B$ represents a resource which can be *combined* with the resource A, and $A \mathrel{-\!\!*} B$ and A together then exchanged for the resource B; this is called *applying* the magic wand. For our purposes, we use A to represent the resources that are given up by the expiring borrow, and B to represent those of the borrowed-from place; the magic wand thus abstractly represents the remainder. In particular, assertion A and B each encode Rust types for which we already have translations. Magic wands are also supported by Viper [Schwerhoff and Summers 2015].

For lender functions, we generate Viper postconditions to be a conjunction of: (1) the Viper embedding of the PCS for the places returned by the function, (2) the translation of any user postconditions regarding these places, and (3) a magic wand $A \mathrel{-\!\!*} B$, where A is the same assertion as (1), and B is the Viper embedding of the capabilities for the parameter from which the returned reference was borrowed. For example, for the `nth_point` function of Fig. 3, we generate[3]:

```
method nth_point(r: Ref, n: Ref) returns (res: Ref)
  requires RefMutRoute(r) && i32(n) && 0 <= n.val && n.val < length(r)
  ensures  RefMutPoint(res) &&
           res.val.x == old(nth_x(r, n)) &&
           (RefMutPoint(res) --* RefMutRoute(r))
```

where `RefMutPoint` and `RefMutRoute` are the predicates generated for mutable references to structs `Point` and `Route`, respectively. The magic wand in the example represents both the partial capability for `r` and the promise that this partial capability can be combined with the capability currently associated with `res` to obtain those originally associated with `r`; by applying the magic wand (at call site), we make use of this promise to restore full capabilities for `r`. As we will show in Sec. 6, the ability to connect the capabilities returned on expiry with the capabilities of the borrowed-from data structure is also essential for adding functional specification to lender functions.

4.3 Automating Proofs with Borrows

Viper supports the magic wand connective, but requires annotations in order to reason about it [Schwerhoff and Summers 2015]. We generate these annotations using our recorded information from Sec. 4.1.

Restoring Capabilities. In our core proof, we generate operations to formally explain how capabilities are restored when borrows expire. Intuitively, we use the recorded borrow information to undo the borrows in an order *opposite* to that in which they were created, using our extracted reborrow relation. Starting from the PCS describing the capabilities just before the borrows expire, we perform the following steps for each borrow. (1) We synthesise any necessary pack/unpack

[3]Viper also requires us to unfold predicates around expressions which require permissions from their bodies; our work generates these annotations too, but we elide them here for readability.

operations to obtain the place capability for the borrower (for instance, the left-hand side of a borrowing assignment); these are encoded in Viper as `fold`/`unfold` operations as explained in Sec. 3.2. (2) We replace this capability with the place capability from which it was borrowed (for instance, the right-hand side of a borrowing assignment). For a direct assignment, this is a no-op in Viper (which already knows the equality of the two locations); for reborrows returned from lender functions, the replacement is encoded by *applying* the corresponding magic wand, directed in Viper via an explicit **`apply`** statement.

Consider for example the program point after the call to `shift_x` at line 56 in Fig. 3. At this point, the borrow `p` expires. Based on the information we record, we know that `p` was returned from the lender function `nth_point`, blocking the function's parameter `r` from being usable, so we apply the wand `RefMutPoint(p) --* RefMutRoute(r)`, which was returned by the Viper encoding of the function call.

Creating Reborrows. When verifying the definition of a function returning a borrow (such as `nth_point` above), our core proof needs to create the required magic wand, which is done in Viper using a `package` statement. This Viper statement must be annotated with a proof of *how*, given *any state* satisfying the wand's left-hand side, one will be able to reassemble the wand's right-hand-side. As a side-effect, the `package` statement consumes any additional resources needed to obtain this right-hand-side (these reflect the remainder capabilities discussed in Sec. 4.2). We generate these proofs automatically, in an analogous way to the explanation of expiring borrows in the previous paragraph. The annotations required to automate our proofs in Viper can be elaborate, but we demonstrate in Sec. 7 that we are consistently able to generate them fully automatically.

We now have the machinery in place to construct a core proof for Rust code that may include mutable borrows and reborrows. We use similar techniques to handle reborrows inside loops (for example, when a Rust reference is used to traverse a recursive heap data structure) in order to generate the required loop invariants at the Viper level completely automatically. In the next section we will explain how to extend our core proof to shared borrows, and in Sec. 6 how to specify and verify functional properties concerning reborrows, e.g. to add specification to lender functions such as the missing postcondition of function `shift_nth_x` in Fig. 3.

5 SHARED BORROWS

Mutable borrows provide temporary exclusive access to a place, but prevent multiple usable aliases. In contrast, Rust's *shared references* (or *shared borrows*) permit multiple references to exist to the same place, or to a place and its sub-places, simultaneously. To ensure the absence of data races and unexpected side effects via aliasing, Rust's type system enforces that the shared parts of the data structure are immutable while at least one such shared reference exists. In this section, we extend the verification technique presented so far to support shared references.

5.1 Read and Write Capabilities

To distinguish between exclusive, mutable access and shared, immutable access, we refine the capabilities associated with a place. We associate *shared capability* with places that store shared borrows, or which are currently borrowed from by a shared borrow, and use *exclusive capability* for all other places (exclusive capabilities correspond to the capabilities used in the previous sections). This refinement also affects the PCS and PCS operations from Sec. 3: we refine place capability sets into maps PCS $\rightarrow$ {exclusive, shared} to specify which capability is associated with each place in a PCS. In the initial PCS of a function, parameters of a shared-reference type (such as `&Point`) are mapped to a read capability, and parameters of other types to a write capability.

Pack and unpack operations (*cf.* Def. 3.2) are extended accordingly: unpacking assigns to all the sub-places the capability of the unpacked place, while packing requires all the sub-places to have the same kind of capability, which is then assigned to the packed place. Recall that PCS operations are applied between statements to reorganise capabilities. In order to go between exclusive and shared capability, we employ two additional PCS operations, called *downgrade* and *upgrade*. These exchange an exclusive place capability for a shared one and vice versa.

When a shared borrow is created by borrowing a place for which exclusive capability was held, this causes a downgrade to shared capability, which is then duplicated for both the borrowed-from place (to make it immutable while the shared borrow exists) and the newly-created borrow. The original exclusive capability of the borrowed-from place is restored (by an upgrade) only when the borrow checker has determined that all shared borrows have expired, and so the place (and its sub-places) can no longer be aliased via shared references.

```
1  // Count the points of `r` inside the rectangle identified by `a` and `b`
2  fn count_inside(r: &Route, a: &Point, b: &Point) -> u32 { /* ... */ }
3
4  // Move the first point until it is unique in `r`
5  fn make_first_unique(r: &mut Route) {
6      let first = &r.current;
7      // r.current.x += 1;
8      let first2 = first;
9      if count_inside(r, first, first2) > 1 {
10         assert!(first.x == r.current.x);
11         r.current.x += 1;
12         make_first_unique(r);
13     }
14 }
```

Fig. 4. An example using shared references. Function `make_first_unique` keeps incrementing the `x`-coordinate of the first point in route `r` until this point's coordinates are unique among all points in `r`. By calling function `count_inside` with shared references to the same point for parameters `a` and `b`, it yields how often that point occurs in `r`.

Function `make_first_unique` in Fig. 4 illustrates the use of shared references. Line 6 creates a shared reference, which downgrades `r` to be temporarily immutable for as long as a shared borrow exists. The type system disallows modifying `r` in line 7 because `r` has a shared capability; in contrast, the assignment in line 11 is permitted because the last shared borrow for `r` expires at line 10, causing an upgrade to exclusive capability for `r`. In the meantime, the assignment in line 8 creates a second shared borrow, which expires after the call to `count_inside`. Since this function takes only shared references, it cannot modify `r` and, therefore, the assertion in line 10 holds.

5.2 Encoding Read Capabilities

Many separation logics support *fractional permissions* [Boyland 2003] to distinguish between read and write access to memory locations. In these logics, a permission can be split into several fractions. A full permission allows (exclusive) write access, whereas any non-zero fraction permits read access. After a full permission has been split into fractions, those fractions can be re-combined to get back the full permission and, thus, write access.

Viper supports fractional permissions as its standard means of expressing read-only access to the heap; we therefore use these to model shared capabilities. However, in order to accurately reflect Rust's type-checking, and fully exploit the information we can extract from the Rust compiler, we

use fractional permissions in a non-standard way. Constructing a standard proof in a fractional-permission logic would require elaborate specifications to keep track of the fractional amounts of permission associated with each shared borrow (which changes as new borrows are created) and to describe how fractional permissions are transferred between different function executions. In particular, we would need to precisely reassemble these fractional amounts to justify restoring write access to a Viper heap location. However, it is not such accounting which should conceptually justify write access: it is the point at which the Rust compiler performs an upgrade to restore an exclusive capability because all involved shared references have expired.

Therefore, we construct our core proof relying on Rust's borrow checker to indicate when a first shared borrow for a place is created (and a downgrade is performed to make the borrowed-from place temporarily immutable) and when the last shared borrow expires (and a corresponding upgrade is performed, so that the place becomes mutable again). In particular, our core proof does not need to add up fractions in order to reassemble a full permission and, thus, the precise fractional amounts associated with shared borrows are irrelevant, as long as we can distinguish between read and write access.

Our encoding uses full permissions for write access (as in the previous sections) and a so-called symbolic read permission [Heule et al. 2013] for read access. A *symbolic read permission* uses a fractional amount that is unspecified, but constrained to satisfy the following properties: (1) it is greater zero and, thus, permits read access, and (2) the sum of all symbolic amounts for any given resource is less than a full permission. The latter property allows one to create additional symbolic read permissions without the risk of ever obtaining (or exceeding) a full permission.

Our encoding maintains the following invariant for each place p whose type is not a shared-reference type: if p is not borrowed from, there exists a full permission for the associated Viper resource; otherwise, if there exist shared borrows for p then p and each of the shared references is associated with a symbolic read permission. To maintain this invariant, we encode operations on shared borrows as follows. When the capability for p is downgraded, we replace the full permission for the corresponding Viper resource with a symbolic read permission (through a Viper **`exhale`** operation) and create a symbolic read permission for the new shared reference (through an **`inhale`** operation). Conversely, when the last shared borrow expires (and a downgrade is performed), we remove its read permission and restore the full permission of the borrowed-from place. In between, when additional shared borrows are created, we simply give them an additional symbolic read permission. This forging of symbolic read permissions is sound because, as we explained above, the sum of all symbolic read permissions is constrained to be less than a full permission. Analogously, we forge a new symbolic read permission when a shared reference p is passed to a function call. This encoding allows the caller to retain its read permission to p and to use it for framing, that is, conclude that the function call cannot change the referenced value.

In the example of Fig. 4, function `make_first_unique` starts out with a full permission to `r` because it is a *mutable* reference. The assignment in line 6 creates the first shared borrow for `r` and, thus, replaces `r`'s full permission with a symbolic read permission. Line 8 creates another shared borrow with another read permission. When calling the function in line 9, the caller retains a read permission to `r` and is therefore able to conclude that the call does not affect the equality `first == r.current`, which was established in line 6 and which implies the assertion in line 10. After this assertion, the last shared borrow of `r` expires, which restores full permission and, thus, enables the assignment in line 11 as well as the subsequent recursive call.

6 PLEDGES FOR MUTABLE BORROWS

It is common for Rust functions to be passed a reference (borrow) as parameter, create a further reference from it (via reborrowing) and to return the new reference to their callers. This idiom

```
#[ensures="after_expiry<result>(
    length(r) == old(length(r)) &&
    nth_x(r, n) == before_expiry<result>(result.x) &&
    forall i: i32 :: (0<=i && i<length(r) && i != n) ==>
        nth_x(r, i) == old(nth_x(r, i))
)"]
fn nth_point(r: &mut Route, n: i32) -> &mut Point { /* ... */ }
```

Fig. 5. An example of our pledge specification feature. The postcondition shown here is the missing third postcondition of `nth_point` from Fig. 3. The first conjunct states that the function does not change the length of the route before creating the borrow, by relating the prestate of a call to `nth_point` to the state in which the `result` borrow expires. Similarly, the third conjunct states that all points other than the n^{th} have unchanged `x`-values. The second conjunct relates the `result` borrow right before the expiry to the rest of the borrowed-from place.

is for instance used in getters such as function `nth_point` in Fig. 3. Callers of such *lender* functions require information about the new borrow as well as the borrowed-from place in order to determine properties of the data structure when the borrow expires. For instance, verifying the three postconditions of `shift_nth_x` in Fig. 3 relies on the fact that the call to `nth_point` does not modify the route `r`; if the call, for instance, removed the first point from the route, none of the postconditions would hold.

Since lender functions (such as `nth_point`) may modify the borrowed-from place before the borrow is created, information about the presence or absence of such side effects needs to be conveyed to callers via the function's postcondition. For shared borrows, this is possible because the borrowed-from place remains usable and, thus, may be accessed in the function's postcondition. However, as we discussed in Sec. 4, returning a *mutable* borrow renders the borrowed-from parameter unusable until the borrow expires. Consequently, referring to the borrowed-from place in the postcondition would violate Rust's type rules and have an unclear semantics as it would introduce aliasing among mutable references. In this section, we introduce pledges, a novel specification construct that lets us specify lender functions. For instance, pledges let us express that `nth_point` does not modify the route `r` (despite having the capabilities to do so, according to its signature).

Pledges. Pledges are assertions that can be used in postconditions of lenders to specify properties of borrowed-from places guaranteed to hold *at the time when the borrow expires*, that is, when the borrowed-from place becomes usable again. This design is compliant with the Rust rules as it avoids referring to unusable places; as we will explain towards the end of the section, it is also essential for modular reasoning.

When a lender returns a mutable borrow, this effectively separates the place for the borrowed-from parameter into a part that can still be accessed through the returned borrow and an unusable remainder. For example, for `nth_point` from Fig. 3, these are the returned point `p` and the remainder of the route `r`. In general (though not in our example), a lender could have modified both parts of the borrowed-from parameter before returning. After returning, the unusable remainder is known to be unchanged until the new borrow expires. However, the part that can be accessed through the returned borrow may get modified by client code, after calling the lender. For instance, `shift_nth_x` modifies the borrowed point `p`. In a modular setting, lenders cannot anticipate how the returned borrow will be modified. Therefore, pledges specify their guarantees *parametrically with* the state of the returned borrow when it expires. For example, for `nth_point` we need a way to explain how `r` will look after `p` expires, irrespective of how the client manipulates the borrowed memory.

Fig. 5 shows the third postcondition for `nth_point` from Fig. 3. It contains a pledge, expressed as argument to the `after_expiry` construct, which is parameterised by the borrow it specifies (here, `result`). The pledge itself expresses three guarantees: (1) the length of the borrowed-from `Route` `r` will be unchanged since the prestate of the call to `nth_point`, (2) the `x`-coordinate of `r`'s n^{th} `Point` will be equal to that of the returned borrow at the time the borrow expires, and (3) all other `x`-coordinates of the `Route` will be unchanged. Note that `nth_point` can guarantee these properties because the remainder of `r` cannot be modified until the borrow expires and because they hold irrespective of the changes made to `result.x` until then. The `before_expiry` notation lets a pledge refer to the borrow right before it expires; it is needed because the borrow is no longer usable after it expires.

This pledge is sufficiently strong to verify the postcondition of `shift_nth_x` in Fig. 3. It provides strong guarantees about the borrowed-from place without constraining how clients may modify the borrow and, thus, enables modular specifications.

Modularity. Our pledges feature respects Rust's typing discipline by expressing properties of the borrowed-from place (`r` in our example) only in states in which Rust allows it to be used. One might be tempted to consider a simpler alternative design which *does allow* such violations for the sake of writing standard postconditions: e.g. allowing `nth_point` to express directly that `r` is unchanged using `forall i: i32 :: (0<=i && i<length(r)) ==> nth_x(r, i) == old(nth_x(r, i))`. Yet, such a postcondition is useless to a caller that subsequently makes modifications via the returned borrow `p`. Since `p` aliases a sub-place of `r`, any such change may affect properties of `r`, for instance, the value of `nth_x(r, i)`. Consequently, we no longer obtain an automatic way to precisely frame such properties. Clients would need to know precisely how to reach `p` from `r` and how the functions `length` and `nth_x` are defined, which would violate information hiding. Moreover, clients would have to prove inductive lemmas to show how changes via `p` affect properties of `r`, losing automation. Our pledges avoid these problems.

Encoding. The encoding of our verification technique to Viper extends naturally to pledges; a pledge is translated as an additional conjunct on the right-hand-side of the magic wand that is created when a mutable borrow is created. Recall from Sec. 4.2 that the right-hand-side indeed represents the state as it will look once the borrow expires; pledges let us complement the resources there with properties of the values stored in the borrowed-from place.

When such a magic wand is packaged (*cf.* Sec. 4.3), Viper proves that the claimed pledge will indeed hold for *any* future state of the borrowed memory, which is necessary to soundly account for any possible changes to the borrow. Viper permits expressions **`old`**`[lhs](e)` on the right-hand-side of a magic wand, which evaluate to *e*'s value in the state in which the wand is applied; this feature enables us to encode our `before_expiry` construct in a straightforward way.

7 IMPLEMENTATION AND EVALUATION

We have implemented our work as a plugin for the Rust compiler, and evaluated it on a wide variety of crates (Rust packages) from the Rust package repository. The evaluation shows that our technique can generate core proofs fully automatically and verify interesting correctness properties without the need for complicated specifications.

7.1 Implementation

We implemented a tool called Prusti as a Rust compiler plugin, usable with Cargo, the official package manager for Rust. Working with Prusti provides a similar experience to existing tools used by Rust developers, such as the Rust linter Clippy [Clippy contributors 2019]. Prusti performs its main work after the type checking pass of the Rust compiler. We extract the compiler's CFG representation

(MIR) along with type and borrow-checker information, construct the corresponding Viper program, and verify it with Viper's symbolic execution verifier; verification results are translated back from Viper to Rust and reported using the Rust compiler's error reporting mechanisms. In addition to proving user specifications, Prusti optionally checks absence of panics and overflows.

Our current tool works on an interesting subset of `safe` Rust, including for instance heap-allocated data and **`Box`** types, shared and mutable borrows, traits, generics, and common use cases of lifetime parameters to functions. We have not yet implemented support for reborrowing inside loops, pure functions returning non-primitive types, and abrupt termination of loop bodies; these restrictions will be lifted in the future (and can typically be worked around by rewriting the program). During the development we built a test suite of more than 300 correct and incorrect Rust programs (annotated with expected verification errors) to check that we model corner cases of Rust's semantics correctly.

To support libraries, our tool provides a `#[trusted]` annotation, allowing us to equip functions with contracts used by callers but *not* checked against the function's implementation.

7.2 Evaluation

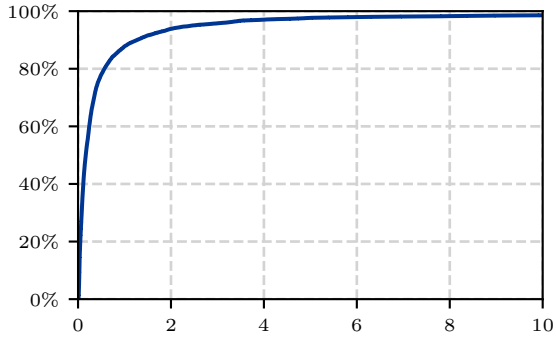
We evaluate our work in three ways: (1) we evaluate the construction of core proofs on all functions from the top 500 Rust crates that fall within our supported language subset; (2) we evaluate the ability to verify simple functional properties by proving the absence of overflows in examples that check for overflow at runtime, to determine whether these runtime checks may ever fail (without any user-provided specifications); (3) we evaluate the use of user-provided specifications by verifying panic freedom and richer functional correctness properties of existing implementations of well-known algorithms. All timings were performed using a clean Ubuntu 18.04.1 installation, on a desktop with a 4-core (8 hyper-threads) Intel i7-2600K 3.40GHz CPU, 32GB of RAM and an SSD disk.

(1) Core Proofs. To test the automation of our core proof construction, we took the 500 most popular Rust community crates [Rust community 2018a], and applied three simple filters: firstly, we discarded any crates (148) which did not compile successfully within 15 minutes using the standard compiler[4] (without our tool); secondly, we filtered all remaining 56,257 functions (top-level, **`impl`** and trait functions) with a simple syntactic check for unsupported language features; thirdly, we manually discarded ten unusually large functions that would have taken more than one minute just for the encoding, due to the large number of local variables used (five implement 4×4 matrix operations; the other five contain huge match expressions with up to 2,000 cases). This left us with 11,791 functions (21% of the total) to evaluate our work on. We re-ran the compiler with Prusti on the unmodified source code of these functions to generate and verify core proofs.

The verification of these 11,791 functions succeeded as expected, without any need for manual intervention. This shows that we generate sufficient annotations to automate the core proof in Viper. These annotations are non-trivial: we generated a total of 1,140,384 lines of Viper code, of which 138,499 are **`fold`**, **`unfold`**, **`package`** or **`apply`** statements to automate the proofs.

We measured how much time is required by Viper to verify each function, reporting results (averaged across three runs) in Fig. 6 (left). We observe that the average verification time per function is 1.2 seconds, that only 0.16 seconds are enough to verify 50% of the functions, and that almost all the functions (98.6%) are verified in less than 10 seconds each. A small fraction of the functions takes more than 10 seconds to verify, for the same reasons as the ten unusually-large functions we discarded (see above). Since the MIR representation used for the encoding is highly

[4] `rustc nightly-2018-06-27`; flags `-Zborrowck=mir -Zpolonius -Znll-facts` and using the reference Polonius algorithm ("Naïve").



Error	Num
assert!(..) might not hold	2
unreachable!(..) might be reachable	1
add with overflow	77
remainder with a divisor of zero	50
remainder with overflow	48
divide by zero	42
divide with overflow	45
multiply with overflow	105
negate with overflow	18
subtract with overflow	71
solver incompleteness	8

Fig. 6. Left: cumulative distribution of the verification time (horizontal axis, in seconds) required for the core proof verification of each of the 11,791 supported functions (177 functions required between 10 and 120 seconds; 11 required between 120 and 888 seconds). Right: distribution of error messages for the overflow freedom evaluation on 520 functions.

unoptimized and uses significantly more (temporary) local variables than the source program, we can reduce this overhead in the future by enabling simple optimisations that the compiler runs in later stages.

(2) Overflow Freedom. We identified 520 functions which potentially raise panics due to overflows or assertions. We re-ran Prusti on these, enabling checks for panics and overflows (again, without specifications). Interestingly, 52 of these functions verified; on manual inspection, this was due to expressions that cannot overflow (e.g. `x-x/4`), or that were guarded by range checks. Since our tool proves soundly that these checks can never fail, one could eliminate them to improve performance without compromising safety.

For each of the remaining 467 functions, Prusti reported a verification error, listed in Fig. 6 (right). Manual inspection showed that these are mostly due to implicit assumptions on argument ranges; our technique makes it possible for developers to make these assumptions explicit as preconditions, and verify them at each call site. In eight cases, Prusti failed to prove that Rust's dynamic overflow checks actually imply that an operation does not overflow. Our inspection revealed that these verification failures are due to the handling of non-linear arithmetic in the underlying Z3 SMT solver. Increasing Viper's timeout for each Z3 query from 10 to 60 seconds results in "divide by zero" verification errors in all eight cases, which is the expected result.

(3) Specifications and Functional Behaviour. In the third part of the evaluation, we investigated the specification and verification of both absence of panics and richer functional properties, using examples from the programming chrestomathy site Rosetta Code [Rosetta Code contributors 2018], a Rust tutorial on linked lists [Rust community 2019], and from Matsakis' blog posts on Rust's language design [Matsakis 2018a,b]. From Rosetta Code, we manually selected a diverse list of eleven examples that either fall into the supported subset of the language or can be adapted without major changes. In order to handle examples using standard library types, we wrote wrappers marked with `#[trusted]` for these types (as explained above); we also rewrote **`for`** loops as **`while`** loops, and restructured some code to avoid **`return`** and **`break`** statements.

The table in Fig. 7 gives an overview of the verified examples (we provide the code including specifications as auxiliary material). Before any manual modifications, the Rosetta Code examples had between 10 to 89 lines of code (excluding blank lines and comments) and between 1 and 6 functions. The average total verification time (averaged over 3 runs) is typically less than 30 seconds, which we consider reasonable for our so far unoptimised encoding and tool. The slowest examples

Example	LOC	#Fns	Spec. LOC	Time (s) All	Time (s) Viper	No Panic	No Overflow	Verified Additional Properties
100 doors	19	2	7	10.9	7.4	✓	✓	
Binary Search (generic)	16	1	2	16.2	12.9	✓	✓	
Heapsort	39	3	18	30.6	26.2	✓	✓	
Knight's tour	89	6	71	127.6	120.2	✓	✓	
Knuth Shuffle	16	2	3	9.5	6.2	✓	✓	
Langton's Ant	58	4	22	16.7	11.8	✓	✓	
Selection Sort (generic)	20	2	8	19.2	15.2	✓	✓	
Ackermann Func.	16	2	17	7.4	4.4	-	×	Correct result
Binary Search (monomorphic)	16	1	29	25.5	21.4	✓	✓	Correct result
Fibonacci Seq.	46	6	26	9.1	5.7	-	-	Correct result
Knapsack Problem/0-1	27	1	86	139.4	131.6	✓	×	Correct computation
Linked List Stack	59	5	60	21.4	16.9	✓	-	Correct behaviour
Selection Sort (monomorphic)	20	2	34	29.6	24.2	✓	✓	Sorted result
Towers of Hanoi	10	2	5	5.9	3.2	-	✓	Correct param. range
Borrow First	7	1	1	6.6	3.6	✓	✓	
Message	13	1	0	7.2	4.2	×	-	

Fig. 7. An overview of the examples verified in the third part of the evaluation. The column "LOC" indicates the number of lines in the unmodified example; "#Fns" is the number of verified functions; "Spec. LOC" is the number of lines used for specification and ghost code; "All Time" indicates the time in seconds required to encode and verify the example; "Viper Time" is just the time needed by the Viper symbolic execution back-end verifier to verify the encoding. "No Panic"/"No Overflow" shows whether we verified absence of panics/overflows ("–" means that the example contains no operations that could panic/unchecked arithmetic). The first two groups of examples are taken from the Rosetta Code website [Rosetta Code contributors 2018], except the "Linked List Stack" example which we took from [Rust community 2019] because it is more complete than the one in Rosetta. The second group differs from the first one in that we verified some functional properties. For example, for the "Ackermann Func." and "Fibonacci Seq." examples, we showed that multiple implementations all compute the correct result. We had to monomorphise "Binary Search" and "Selection Sort" for proving stronger functional properties because Prusti does not yet support intrinsic trait properties such as transitivity of the equivalence operator. The precondition we chose for the Ackermann functions do not prevent overflow and, thus, this aspect could not be verified (indicated by "×"). In "Knapsack Problem/0-1" we verified correctness of all intermediate computations; correctness of the result and absence of overflow would require sum comprehensions, an advanced specification feature not yet supported in Prusti. The two examples in the last group are from Matsakis' blog posts about non-lexical lifetimes in Rust [Matsakis 2018a,b]. For one of them, proving panic freedom failed because the program does not handle all IO errors.

"Knight's tour" and "Knapsack Problem/0-1" take less than two and a half minutes (each of them contains one large function that takes most of the time). In all cases, standard deviations were around 1 second.

For most examples, we verified the absence of panics and overflows, by adding specifications where necessary. In some cases, for example for "Binary Search", this required adding only a simple invariant that the indices are no larger than the vector's length, which allowed the verifier to prove not only the absence of out-of-bounds accesses, but also the absence of overflows. In other cases, for example for "Knight's tour", we had to add ghost code to encode object invariants. The most interesting specification for proving panic-freedom is for "Langton's Ant", which required not only quantifiers to specify an invariant of the grid on which the ant walks, but also a pledge to specify how changes made via borrows affect the invariant of the grid. Via our evaluation, we found a bug in the source code of this example, which causes an integer overflow during execution. We fixed this error by correcting existing boundary checks and types.

For seven examples, we also verified properties that go beyond basic safety. For two of them, we had to monomorphise the generic parameters to integers in order to use integer comparisons instead of a trait function. Functional correctness of the binary search example initially failed to verify; closer inspection revealed an off-by-one bug in the source code (a fixed version verifies with our tool). We encode other properties such as sortedness ("Selection Sort"), functional correctness of recursive and iterative implementations ("Fibonacci Seq." and "Ackermann Func."), functional correctness of a data structure ("Linked List Stack"), correctness of intermediate computations ("Knapsack Problem/0-1"), and validity of parameter values in function calls ("Towers of Hanoi").

These seven examples require on average 1.3 lines of annotation per line of code. While this overhead is not negligible, it is lower than the overhead required by existing verifiers for heap-manipulating programs. Moreover, our annotations are conceptually much simpler since they are expressed in terms of Rust expressions rather than complex program logics. Another core advantage of our approach is that the user is not forced to provide all of them from the beginning, but can add them gradually to strengthen the verified properties. For instance, proving safety for "Binary Search" requires only two lines of annotations. To additionally prove that the returned index is correct if **`Some`** is returned, the user needs to add two additional straightforward assertions. Finally, proving correctness for the case that **`None`** is returned is slightly more involved because it requires to write a quantifier that expresses that the vector is sorted. Nevertheless, none of these assertions expose the complexity of program logics for concurrent, heap-manipulating programs.

We also evaluated our tool on two examples from Matsakis's blog [Matsakis 2018a,b], designed to illustrate difficult borrowing patterns. The support for the first example was added to stable Rust only recently, while the second one still requires a nightly-build version of Rust. Both examples are already supported by our tool (using the corresponding new borrow checker implementation).

8 RELATED WORK

Capability-Based Type Systems. Many other type systems can also be understood to associate capabilities with reference types [Boyland et al. 2001]. Some extend pre-existing languages (e.g. Sing♯ [Barnett et al. 2011], C♯ [Gordon et al. 2012] and Scala actors [Haller and Odersky 2010]); more recently, several programming languages have built these in (e.g. Pony [Clebsch et al. 2015], AEminium [Stork et al. 2014], and Rust itself [Matsakis and Klock II 2014]). Such built-in type systems are exploitable by the compiler: e.g. for memory management in Rust, or to enable the distributed garbage collection in Pony. While these systems provide programmers with stronger guarantees than traditional type safety, *functional correctness* of programs cannot be expressed: our work shows how to layer such verification concerns on top, while exploiting the benefits provided by the type system.

Type Systems for Verification. Liquid Types [Rondon et al. 2008] equip types with logical qualifiers prescribing value properties; their extension to Alias Refinement Types [Bakst and Jhala 2016] applies to mutable heap data structures. Type checking is decidable, and loop invariants can be inferred. Unlike our work (*cf.* Secs. 4 and 6), there is no support for references (reborrows) which persist beyond the function calls or loops they are created in.

SYMPLAR [Bierhoff 2011] targets formal verification for Java, employing a notion of permissions to separate reasoning about aliases from verification conditions concerning values. Like our work, user-specification is at the level of the programming language. A planned addition to SPARK (a subset of Ada designed for formal verification) will add pointer support [Maalej et al. 2018], using a type system similar to Rust. In both systems, returning reborrowed references is not supported.

Rust Verification Tools. CRUST [Toman et al. 2015], a recent adaptation of SMACK [Baranowski et al. 2018], and Lindner *et al.* [Lindner et al. 2018] provide *bounded* verification tools for Rust

(including unsafe code); these tools allow user checks to be added as Rust expressions. These tools work on C/LLVM code where Rust's type information is absent. By contrast, we exploit this information for modular *unbounded* (sound) verification, and support richer functional specifications via `old` expressions and pledges.

Ullrich [Ullrich 2016] encodes safe Rust programs into functional programs, to be interactively verified in Lean [de Moura et al. 2015]. Reborrows are supported via lenses [Foster et al. 2005]. Recent work at Galois similarly reduces reasoning about a subset of safe Rust to proofs about functional programs in Saw [Dockins et al. 2016]. In contrast to these works, our technique does not require the manual construction of proofs or verifier directives; in addition, our underlying separation logic formalism will provide a suitable (imperative-style) model for an extension to unsafe code in the future.

As a general point, we believe our implementation to be the first verification technology so far to operate directly on the Rust compiler's analysis results and representations of source programs; there is no gap between the Rust programs and notions and the starting point for our work.

Rust Semantics and Formalisations. A number of formalisms for subsets of Rust have been designed, focusing on type soundness results [Kan et al. 2018; Reed 2015; Wang et al. 2018; Weiss et al. 2018]. It would be interesting to compare these formal models with the PCS/borrow summaries that our work produces from the compiler.

Rustbelt [Jung et al. 2017] provides a formalisation aimed at proving *unsafe* library implementations to encapsulate their unsafe behaviour, and defining formally what this notion should mean for Rust. As explained in the introduction, the goals and contributions of our work are very different; we do not address Rust semantics, and our technique enables users to be shielded from the complexity of formal logics capable of expressing such semantics. There are also important technical differences in the underlying logics: Rustbelt's handling of borrow expiry supports more cases of borrows (even in unsafe code), but does not express a direct connection between the *contents* of memory returned by these borrows and the resulting contents of borrowed-from memory (handled by magic wand assertions in our work); without substantial additional ghost code, we believe that our pledges specifications cannot be directly encoded in Rustbelt's logic.

9 CONCLUSIONS

We presented a new specification and verification technique for Rust, leveraging the guarantees provided by the language's type system, and synthesising from these automatic core proofs in a logic akin to separation logic. By providing specifications at the level of abstraction of the Rust language, programmers can extend this core proof to verify rich functional properties. Verification is performed via an automatic translation to the Viper infrastructure. A key virtue of our technique is that it does not expose the complexity of the underlying verification logic; programmers work exclusively on the level of Rust programs, which facilitates adoption. Our work is implemented and freely available. Our evaluation shows that we can reliably automatically construct core proofs for real-world Rust code and verify functional correctness properties without resorting to complex program logics.

Our main goal for future work is to extend the subset of Rust supported by our technique and tool, in particular, to add iterators, closures, and structures with lifetime parameters; these extensions will broaden the applicability of our technique to a wider range of real-world code. We also plan to add support for certain classes of *unsafe* code.

REFERENCES

Martìn Abadi and Marcelo Fiore. 1996. Syntactic Considerations on Recursive Types. *Proceedings - Symposium on Logic in Computer Science*, 242 – 252. https://doi.org/10.1109/LICS.1996.561324

Alexander Bakst and Ranjit Jhala. 2016. Predicate Abstraction for Linked Data Structures. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583 (VMCAI 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 65–84. https://doi.org/10.1007/978-3-662-49122-5_3

Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 528–535.

Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (June 2011), 81–91. https://doi.org/10.1145/1953122.1953145

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL) (LIPIcs)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl, 1:1–1:12.

Kevin Bierhoff. 2011. Automated Program Verification Made SYMPLAR: Symbolic Permissions for Lightweight Automated Reasoning. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 19–32. https://doi.org/10.1145/2048237.2048242

John Boyland. 2003. Checking interference with fractional permissions. *Static Analysis* (2003), 1075–1075.

John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001 — Object-Oriented Programming, 15th European Conference* (Budapest, Hungary, June 18–22) *(Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.). Springer, Berlin, Heidelberg, New York, 2–27.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

Clippy contributors. 2019. Clippy. https://github.com/rust-lang/rust-clippy Accessed April 4, 2019.

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 23–42.

Coq Team. 2014. The Coq Proof Assistant Reference Manual. http://coq.inria.fr.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 50–63. https://doi.org/10.1145/301618.301641

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (system description). In *Automated Deduction - CADE-25, 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*.

Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 56–72.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem. *SIGPLAN Not.* 40, 1 (Jan. 2005), 233–246. https://doi.org/10.1145/1047659.1040325

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. *SIGPLAN Not.* 47, 10 (Oct. 2012), 21–40. https://doi.org/10.1145/2398857.2384619

Philipp Haller and Martin Odersky. 2010. *Capabilities for Uniqueness and Borrowing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification.. In *OSDI*. 165–181.

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Vol. 7737. Springer-Verlag, 315–334.

Cliff B. Jones. 1983. Specification and design of (parallel) programs. In *IFIP Congress*. 321–332.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* 50, 1 (2015), 637–650.

Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 2018. K-Rust: An Executable Formal Semantics for Rust. *CoRR* abs/1804.07608 (2018). arXiv:1804.07608 http://arxiv.org/abs/1804.07608

Ioannis T. Kassios. 2011. The Dynamic Frames Theory. *Formal Aspects of Computing* 23, 3 (2011), 267–289.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. 2011. *JML Reference Manual*. http://www.jmlspecs.org/.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of LPAR'10*. Springer-Verlag, Berlin, Heidelberg, 348–370. http://dl.acm.org/citation.cfm?id=1939141.1939161

K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, M. Odersky (Ed.), Vol. 3086. Springer-Verlag, 491–516.

K. Rustan M. Leino and Greg Nelson. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.* 24, 5 (2002), 491–553.

Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. 108–114. https://doi.org/10.1109/INDIN.2018.8471992

Maroua Maalej, Tucker Taft, and Yannick Moy. 2018. Safe Dynamic Memory Management in Ada and SPARK. (2018).

Nicholas D. Matsakis. 2018a. MIR-based borrow check (NLL) status update. http://smallcultfollowing.com/babysteps/blog/2018/06/15/mir-based-borrow-check-nll-status-update Accessed April 4, 2019.

Nicholas D. Matsakis. 2018b. MIR-based borrowck is almost here. http://smallcultfollowing.com/babysteps/blog/2018/10/31/mir-based-borrowck-is-almost-here/ Accessed April 4, 2019.

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

Bertrand Meyer. 1992. Design by Contract. In *Advances in object-oriented software engineering*, Dino Mandrioli and Bertrand Meyer (Eds.). Prentice Hall, 1–50.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.), Vol. 9583. Springer-Verlag, 41–62.

Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *Concurrency Theory (CONCUR) (Lecture Notes in Computer Science)*, Philippa Gardner and Nobuko Yoshida (Eds.), Vol. 3170. Springer, 49–67.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of CSL'01*. Springer-Verlag, London, UK, UK, 1–19. http://dl.acm.org/citation.cfm?id=647851.737404

Susan Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (May 1976), 279–285.

Matthew J. Parkinson and Gavin Bierman. 2005. Separation logic and abstraction. In *POPL*, Jens Palsberg and Martìn Abadi (Eds.). ACM, 247–258.

Matthew J. Parkinson and Alexander J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3:01 (2012), 1–54.

Eric W. Reed. 2015. Patina: A Formalization of the Rust Programming Language.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society Press.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. *SIGPLAN Not.* 43, 6 (June 2008), 159–169. https://doi.org/10.1145/1379022.1375602

Rosetta Code contributors. 2018. Rosetta Code. https://rosettacode.org/wiki/Category:Rust Accessed November 5, 2018.

Rust community. 2017. Non-Lexical Lifetimes RFC. https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md Accessed November 4, 2018.

Rust community. 2018a. The Rust community's crate registry. https://crates.io Downloaded on November 2, 2018.

Rust community. 2018b. Rust: The Reference — Place Expressions and Value Expressions. https://doc.rust-lang.org/reference/expressions.html#place-expressions-and-value-expressions Accessed November 4, 2018.

Rust community. 2019. Learn Rust by writing Entirely Too Many Linked Lists. https://rust-unofficial.github.io/too-many-lists/first-final.html Accessed April 4, 2019.

Rust contributors. 2019a. The Polonius Reference Implementation for the Rust Borrow-Checker. https://github.com/rust-lang/polonius Accessed April 4, 2019.

Rust contributors. 2019b. The Rustonomicon: Working with Unsafe. https://doc.rust-lang.org/nomicon/working-with-unsafe.html Accessed April 4, 2019.

Malte Schwerhoff and Alexander J. Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier. In *ECOOP (LIPIcs)*, J. T. Boyland (Ed.), Vol. 37. Schloss Dagstuhl, 614–638.

Jan Smans, Bart Jacobs, and Frank Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP (LNCS)*, Vol. 5653. Springer, 148–172.

Jan Smans, Bart Jacobs, and Frank Piessens. 2010. Heap-Dependent Expressions in Separation Logic. In *FMOODS (LNCS)*, John Hatcliff and Elena Zucca (Eds.), Vol. 6117. Springer, 170–185.

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 2:1–2:58.

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. 2014. AEMinium: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 2 (March 2014), 42 pages. https://doi.org/10.1145/2543920

Alexander J. Summers and Sophia Drossopoulou. 2013. A Formal Semantics for Isorecursive and Equirecursive State Abstractions. In *ECOOP (LNCS)*, Vol. 7920. Springer, 129–153.

John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: A bounded verifier for rust (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 75–80.

Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's thesis. Karlsruhe Institute of Technology.

Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. *CoRR* abs/1804.10806 (2018). arXiv:1804.10806 http://arxiv.org/abs/1804.10806

Aaron Weiss, Daniel Patterson, and Amal Ahmed. 2018. Rust Distilled: An Expressive Tower of Languages. *arXiv preprint arXiv:1806.02693* (2018).

A ENCODING TO VIPER

In this section we will show step-by-step how the method `force_inc` provided in Fig. 8 is encoded to Viper. Some technical details are simplified, because the compiler internally translates the source program to MIR: a CFG-based representation, much more verbose than the original program. At a high level, Prusti encodes each Rust function to a Viper method, each Rust *pure* function to a Viper function, and each Rust type to a Viper predicate.

A.1 Encoding of functions

The encoding of functions is structured in three parts:

(1) generation of **pack** and **unpack** operations;
(2) encoding of the function signature;
(3) encoding of function's statements.

Pack and unpack. The technique described in Sec. 3 is used to annotate the MIR representation with **pack** and **unpack** operations. In the case of `force_inc` the generated operations are shown as comments in the source program of Fig. 8. The first three **unpack** are required to access `r.current.x`; then at lines 22-23 further unpack are needed to access `*r.rest.0`, due to the (implicit) initialization of `rest` at line 25. Note that one **unpack** needs to know the variant of `r.rest`, which inside the branch the compiler knows to be **`Some`**. At the end of the branch, lines 29-30, the **pack** operations are needed to unify the PCS with the one coming from the (empty) **`else`** branch, in which no **unpack** was ever performed. Note that in this case joining the two PCSs is only possible by using **pack** operations, because line 22 used a branch-specific information. Finally, at lines 33-35 the **pack** operations are used to restore the initial fully-packed state of `r`: the only argument of type reference, whose capability needs to be transferred back to the caller.

Pre- and postcondition. The signature of the function is then used to generate the signature of the Viper method, in which each Rust argument is encoded as a Viper reference. The type of the arguments is used to encode the first **`inhale`** in the Viper method, which encodes the permissions of the precondition — that is, the capabilities transferred to the function. Similarly, the last **`exhale`** statement encodes the permissions of the postcondition — that is, the capabilities that the function gives back to the caller. In Fig. 8, only the capabilities of `r` go back to the caller, because it is the only argument of type reference. When specified, the functional specification of the precondition is conjoined to the expression of the first **`inhale`**, and the postcondition to the expression of the last **`exhale`**, as in the case of `force_inc`.

Statements. Each statement is then encoded independently. Considering Fig. 8:

(1) At *unpack-1* and *unpack-2* the **unpack** statements are encoded as Viper's **`unfold`**. Similarly, at *pack-1* and *pack-2* the **pack** statements are encoded as Viper's **`fold`**. Since which fields compose the enumeration `r.rest` depends on the value of the discriminant, the related **pack** and **unpack** PCS operations depend as well on the variant of the enumeration — in this case **`Some`**. The **pack** operations at *unpack-2* are needed to join with the PCS of the (empty) else branch.
(2) At *pure-call* an assignment and call of a function marked as **`pure`** is encoded with a corresponding Viper assignment and function call. Since the Viper encoding uses one more indirection than Rust, any statement that needs to access the value of a Rust type is wrapped in pairs of **`unfold`**-**`fold`** statements; in this case to access the last field of `min_x.val_i32` and `r.val_ref.current.x.val_i32`.

Fig. 8. An encoding of `force_inc` to Viper. The function is augmented with **pack** and **unpack** operations, and is then translated to the Viper method shown on the right. The encoding of the postcondition `<post>` is in Fig. 10. Note that Rust implicitly dereferences `r` when accessing its fields.

(3) The **`if let`** construct is encoded as a lookup of the value of the discriminant, used immediately after as guard in the branching. The **`unfolding`** expression acts similarly to a pair of **`unfold`**-**`fold`** statements, temporarily unfolding `r.val_ref.rest` to access the `discriminant` field.
(4) At *assign* the allocation and initialization of the local variable `q` is encoded using a Viper local variable and an assignment that sets its referenced object. The predicate instance that encodes its type, `RefMutRoute`, is obtained in three steps: first the variable obtains at its allocation the access predicate for the `val_ref` field; then, the predicate instance for the referenced object is obtained from the assignment; finally, a **`fold`** statement packs the wanted predicate.
(5) At *copy* and *call* a call of a non-pure function is encoded. Initially, a temporary local variable `arg2` is introduced to encode the copy of the second argument. Then, at *call* the precondition of the called function is exhaled and its postcondition inhaled. The label `call_pre` is used in the encoding of expressions of the postcondition that refer to the state of the precondition.

A.2 Encoding of types

To verify the example in Fig. 9 it is necessary to encode in Viper (as already described in Sec. 3) all the types used in the program: **`i32`**, `Point`, **`Box`**<`Point`>, **`Option`**<**`Box`**<`Point`>>, `Route`, and **`&mut`** `Route`.

Primitive types. The encoding of **`i32`** consists of a Viper reference with a field of type **`Int`**, which stores the value of the type.

Structures and enumerations. The encoding of Rust structures, like `Point` and `Route`, follows the intuition that a structure is just a set of fields. Indeed, the predicate of each structure encodes the permission to access the fields, plus the type of each field.

Rust enumerations are a generalization of structures. The main difference with structures is that enumerations need to store the *discriminant*: a tag that identifies which variant of the enumeration is actually stored in memory. The value of the discriminant dictates which fields compose an enumeration, and this aspect is encoded using implications in the predicate, as can be seen in the encoding of **`Option`**<**`Box`**<`Point`>>.

Mutable references and boxes. Since in Rust it is possible to reference both local variables and heap allocated structures, the Viper encoding needs a uniform way to identify the referenced memory location. For this reason, we decided to model all Rust values as Viper objects, such that each Viper reference models a Rust memory location. The encoding of Rust references like **`&mut`** `Route`, thus, consists in a Viper reference with a field that stores the referenced location, which holds an instance of the corresponding predicate.

Similar requirements need to be satisfied for the encoding of boxes, already presented in Sec. 3, and the resulting encoding is identical to the encoding of references.

A.3 Encoding of pure functions and specifications

The encoding of Rust functions marked as **`pure`** consists in the construction of a single Viper expression that defines the value returned by the Rust function. The computed expression is then used as body in the definition of a Viper function. Primitive types, in the case of pure functions, are directly encoded with Viper's **`Int`** and **`Bool`** types. The resulting encoding of `max` is almost identical to its Rust version:

```
function max(a: Int, b: Int): Int { a < b ? b : a }
```

The same technique is used to encode the specifications defined in the **`requires`**, **`ensures`** and **`invariant`** attributes. Valid Rust code contained in the specifications is converted to an equivalent Viper expression, with the only difference that `old(..)` expressions need a Viper label that specifies

to which state they refer to. For this reason, **label** statements are generated in Viper to identify with pre the program point in which the precondition holds, and with other labels the program point before each function call. Missing preconditions or postconditions are considered to be **true** by default.

Continuing the example of Fig. 8, the encoding of a possible postcondition for `force_inc` is shown in Fig. 10.

B PCS TRANSFORMATIONS

As explained in Sec. 3, each Rust statement may require certain capabilities to be in the PCS and then transform the PCS. The transformation can be defined as a function that given an input PCS computes the output PCS, corresponding to the post-state of the statement. In this appendix we provide additional details with an example.

Definition B.1. The *prefixes* of a place p is the set of places P_p containing p and all places that are the sub-place of an element of P_p. The *proper prefixes* of a place p is the set of the prefixes of p, excluding p itself. The *extensions* of a place p is the set of places E_p containing p and all the places p' that have a prefix in E_p. The *proper extensions* of a place p is the set of the extensions of p, excluding p itself.

Consider the move assignment `rhs = lhs`, where `lhs` and `rhs` are places. The requirement of this statement is that the input PCS must contain `rhs`. The output PCS is then computed from the input PCS by performing the following operations:

(1) remove `rhs`;
(2) remove all proper extensions of `lhs`;
(3) unpack proper prefixes of `lhs`, if any, until `lhs` is obtained;
(4) add `lhs`.

The first step is needed to mark that `rhs` is moved, thus unusable. The following two steps are needed to bring the PCS in a state in which `lhs` and `rhs` have the same capability. If `lhs` can not be reached that way, the last step adds it. This happens e.g. when initializing new variables or moved-out places.

Example. We can see in Fig. 11 an example with two assignments, of which the first is an implicit statement generated by the **if let** construct. Starting from the top, **if let** requires `curr.next` to be in the PCS, so the unpack at line 8 is generated.

The first assignment has `*curr.next.0` on the right-hand-side, so the requirement is to have it in the PCS. This is fulfilled by lines 13 and 15, which make use of the type-checker information regarding the variant of `curr.next`. The assignment then transforms the PCS using steps (1) and (4), introducing for the first time `tail` in the PCS.

The requirement of the second assignment is already fulfilled by the PCS, so no **pack**/**unpack** is needed. The statement then uses steps (1), (2) and (4) to transform the PCS. Note that in the resulting PCS there is no mention of `tail`, which has been moved-out; `curr` is fully packed, meaning that it does not contain moved-out subfields.

After the assignments, the two branches of **if let** need to be joined, but they have different PCS. The conflict is thus resolved by unpacking `curr` in the first branch. An alternative solution would be to pack `curr` in the **else** branch.

The final PCS has no special requirements, since no expression is returned to the caller and no parameter is passed by reference.

```
predicate i32(self: Ref) {
  acc(self.val_i32)
}

predicate Point(self: Ref) {
  acc(self.x) && i32(self.x) &&
  acc(self.y) && i32(self.y)
}

predicate Route(self: Ref) {
  acc(self.current) &&
  Point(self.current) &&
  acc(self.rest) &&
  OptionBoxPoint(self.rest)
}

// Encoding of type `Box<Point>`
predicate BoxPoint(self: Ref) {
  acc(self.val_ref) &&
  Point(self.val_ref)
}

// Encoding of type `Option<Box<Point>>`
predicate OptionBoxPoint(self: Ref) {
  acc(self.discriminant) &&
  0 <= self.discriminant &&
  self.discriminant <= 1 &&

  // Variant 0: `None`
  (self.discriminant == 0 ==> true) &&

  // Variant 1: `Some(Box<Point>)`
  (self.discriminant == 1 ==>
    acc(self.elt0) &&
    BoxPoint(self.elt0)
  )
}

// Encoding of type `&mut Route`
predicate RefMutRoute(self: Ref) {
  acc(self.val_ref) &&
  Route(self.val_ref)
}
```

Fig. 9. Encoding of the types used in Fig. 8. All the fields are of type **Ref**, except for `discriminant` and `val_i32` which are **Int**.

```
#[ensures="length(r) == old(length(r))"]
#[ensures="r.current.x == max(old(r.current.x), min_x)"]
#[ensures="r.current.y == old(r.current.y)"]
#[ensures="forall i: i32 :: (1 <= i && i < length(r)) ==>
    nth_x(r, i) == max(nth_x(r, i - 1), old(nth_x(r, i)))"]
fn force_inc(r: &mut Route, min_x: i32) { /* ... */ }
```

```
method force_inc(r: Ref, min_x: Ref) {
    // ...
    exhale RefMutRoute(r) &&
        length(r) == old[pre](length(r)) &&
        (unfolding ... r.val_ref.current.x.val_i32) == max(
            old[pre](unfolding ... r.val_ref.current.x.val_i32),
            old[pre](unfolding ... min_x.val_i32)
        ) &&
        (unfolding ... r.val_ref.current.y.val_i32) ==
            old[pre](unfolding ... r.val_ref.current.y.val_i32) &&
        forall i: Int :: (1 <= i && i < length(r)) ==>
            nth_x(r, i) == max(nth_x(r, i - 1), old[pre](nth_x(r, i)))
}
```

Fig. 10. A possible postcondition for method `force_inc` from Fig. 8, with an encoding as Viper expression. `min_x` is implicitly evaluated in the `pre` state, because it is the only state in which the caller can see it.

```
1  struct List {
2      v: i32,
3      next: Option<Box<List>>
4  }
5
6  fn foo(mut curr: List) {
7      // PCS: { curr }
8      unpack curr;
9      // PCS: { curr.v, curr.next }
10
11     if let Some(box mut tail) = curr.next {
12         // PCS: { curr.v, curr.next }
13         unpack curr.next as Some;
14         // PCS: { curr.v, curr.next.0 }
15         unpack curr.next.0;
16         // PCS: { curr.v, *curr.next.0 }
17
18         // Requires: *curr.next.0
19         // Input PCS: { curr.v, *curr.next.0 }
20         let mut tail = *curr.next.0; // Implicit
21         // Output PCS: { curr.v, tail }
22
23         // Requires: tail
24         // Input PCS: { curr.v, tail }
25         curr = tail;
26         // Output PCS: { curr }
27
28         // PCS: { curr }
29         unpack curr;
30         // PCS: { curr.v, curr.next }
31     } else {
32         // PCS: { curr.v, curr.next }
33     }
34
35     // PCS: { curr.v, curr.next }
36 }
```

Fig. 11. Example of Rust program annotated with **pack**/**unpack** operations and PCS for each program point.