ETH zürich

ETH Library

Leveraging rust types for modular specification and verification

Working Paper

Author(s):
Astrauskas, Vytautas; Müller, Peter; Poli, Federico; Summers, Alexander J.

Publication date:
2018-12

Permanent link:
https://doi.org/10.3929/ethz-b-000311092

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Leveraging Rust Types for Modular Specification and Verification

Vytautas Astrauskas
Department of Computer Science
ETH Zurich, Switzerland
vytautas.astrauskas@inf.ethz.ch

Peter Müller
Department of Computer Science
ETH Zurich, Switzerland
peter.mueller@inf.ethz.ch

Federico Poli
Department of Computer Science
ETH Zurich, Switzerland
federico.poli@inf.ethz.ch

Alexander J. Summers
Department of Computer Science
ETH Zurich, Switzerland
alexander.summers@inf.ethz.ch

Abstract

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. Going beyond memory safety, for instance, to guarantee the absence of assertion failures or functional correctness, requires static program verification. Formal verification of system software 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 a 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. 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. Crucially, our proofs are constructed and checked automatically; users of our work never work with the underlying formal logic. Users can add specifications at the abstraction level of Rust expressions; we show how to interweave these to extend our core proof to prove modularly whether these specifications are correct.

We have implemented our technique for a subset of Rust; our initial evaluation on two thousand functions from widely-used Rust crates demonstrates its effectiveness.

1 Introduction

Producing reliable systems 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 a *capability* [9] with each mutable memory location. At any time, each capability is held exclusively by at most one executing function call: only that code may access the memory location. 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 [6, 21, 22, 29], 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 [41, 47], dynamic frames [28, 31], or object ownership [11]. 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. 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. 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. Crucially, our proofs are constructed and checked automatically; users of our work never work with the underlying formal logic. Users can add specifications at the abstraction level of Rust expressions; we show how to interweave these and extend our core proof to prove modularly and automatically whether these user specifications are correct. 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.

ETH Research Collection, Tech. Report, 2018
16th December 2018.

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 for enabling modular proofs of Rust programs which pass *borrowed references* in and out of function calls.
3. We define a verification technique that encodes both capability information and user-provided assertions into the implicit dynamic frames logic [52], a close relative of separation logic [45].
4. We automate our verification technique by constructing a translation from the Rust program and specifications into the Viper intermediate verification language [40]. Our translation generates correct specifications and synthesises all necessary auxiliary annotations needed for proof checking to be 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. Our plugin is available [2]; we will also submit it as an artifact.

Unsafe Code and Rustbelt Rust's type system enforces strong rules, but provides and escape hatch for when these are too restrictive: code blocks and functions can be declared *unsafe*, weakening the compiler's checks, but correspondingly risking the guarantees these provide. Unsafe code *should* be encapsulated by libraries such that client code cannot observe its usage [57]. The ongoing *Rustbelt* project [25] 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 first paper, but present a verification technique enabling user-specifications at a high level of abstraction and automatic proofs; Rustbelt verification uses an advanced separation logic based on Iris [26], for which proofs are interactive (in Coq [54]), and constructed by experts (See also Sec. 7).

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 borrowed references; Sec. 4 extends our techniques to handle these. Sec. 5 introduces pledges, used to attach functional specifications to borrows. We describe and evaluate our implementation in Sec. 6, discuss related work in Sec. 7, and conclude in Sec. 8.

```
struct Point {
  x: i32, y: i32
}

#[ensures="p.x == old(p.x) + s"]
#[ensures="p.y == old(p.y)"]
fn shift_x(p: &mut Point, s: i32) {
  p.x = p.x + s
}

fn compress(mut segm: (Box<Point>, Box<Point>))
                    -> (Box<Point>, Box<Point>) {
  let diff = (*segm.0).x - (*segm.1).x;
  shift_x(&mut segm.1, diff + 1);
  assert!((*segm.0).x < (*segm.1).x);
  segm
}
```

Figure 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`.

.

2 Motivating Example

In this section, we illustrate our approach from a programmer's perspective. Details of the technique 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 borrowed reference. 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. 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 borrowed reference to `segm.1`) as the compiler infers that the borrow *expires* after the call, restoring capabilities to the borrowed-from `segm.1`.

Correctness Arguments Proving that the assertion holds for all calls to `compress` requires the following properties, where 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 [28, 32, 47]; 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 [23, 43] or explicit proofs of race freedom [41], 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 Rust annotation. Our specification language is based on Rust boolean expressions, extended with few (but powerful) additional constructs; here, the `old` construct [30] 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-states of a call. The second postcondition of `shift_x` is not needed to verify the assertion, but is likely required by other clients. 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 [39], 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 the re-verification effort during maintenance.

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.

The starting point of our verification process is to encode the capability information of a program to obtain a logical proof of memory safety. It is crucial that this encoding 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 Rust provides. The core proof by itself guarantees the same properties as the type system. Its main value is that it provides the foundation to verify stronger properties, such as the correctness of user-provided assertions, and the absence of arithmetic overflows and various kinds of exceptions (called *panics* in Rust), including **`assert!`** failures. Constructing the core proof is relatively easy for simple code like our example. However, handling complex forms of (re)borrowing and synthesising auxiliary annotations to automate the proof search is challenging, 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 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. 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 Sec. 4. We present the capability information that is needed to construct a proof of memory safety, explain how we construct this proof in 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: **`struct`** and **`enum`** types, move and copy assignments, heap-allocated data (including **`Box`** types), mutable borrows (including reborrowing), loops, and function calls, including common use-cases of lifetime parameters to functions. Commonly-used Rust features which fall outside of our subset include: shared borrows, traits, generics, and lifetime parameters to **`struct`** types, as well as *unsafe* code. Supporting those features is 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,

```
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>))
8                      -> (Box<Point>, Box<Point>) {
9    let mut end = segm.1; // move assignment
10   // segm.1 is now inaccessible
11   let diff = (*segm.0).x - (*end).x;
12   end = shift_x(end, diff + 1);
13   segm.1 = end;
14   // end is now inaccessible
15   assert!((*segm.0).x < (*segm.1).x);
16   segm
17 }
```

Figure 2. A variation of the example from Fig. 1 that uses move assignments instead of borrowing. The move assignment in line 9 removes the capability for `sgem.1` until it is restored in line 13, which prevents accesses in between. In particular, omitting line 13 would cause a compiler error, since it would not be possible to assemble the full capabilities required by the return type signature.

but also which *places* (Rust's terminology for expressions denoting memory locations [55]) 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 execution of a function. 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 its deallocation time. 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 indicate the owner of a value; 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 later. The assignment in line 9 is a *move assignment*, which transfers ownership form `sgem.1` to `end`, making `sgem.1` inaccessible. Similarly, the call to `shift_x` transfers ownership from `end` to parameter `p`. Ownership is restored to `end` when the function terminates, and to `segm.1` in line 13. The subsequent assertion holds for reasons similar to those outlined in the previous section. 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. 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 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: e.g. throughout a function body, struct-typed variables retain the same Rust type, but capabilities to their fields vary as they are borrowed or moved. We therefore 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. Rust's type checker also computes such information, but using representations which are not exposed and, for borrowed references, not as detailed as necessary for our work.

Definition 3.1 (Place Capability Sets)**.** *Places*, ranged over by p, are expressions defined by the following grammar: $p ::= x \mid p.f \mid (*p)$. A place p' is a *sub-place* of p if it is additional field/pointer dereferences starting from p.
A *place capability set* (PCS) is a finite set of places.

The initial PCS for a function contains exactly 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 9 requires the initial PCS to contain `segm.1`, and transforms the PCS from {`segm.0`,`segm.1`} to {`segm.0`,`end`}, reflecting the move of capabilities. This move assignment is permitted because place capabilities also imply capabilities on all sub-places, that is, the type checker may *unpack* the capability for `segm` into {`segm.0`,`segm.1`}. Type checking may perform the following operations on 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 $p.f_n$ (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. Remove is used at join points

in the control flow if the joined paths provided different capabilities; unpack is needed to access fields of a struct, and pack is used when the entire struct is passed as argument or result such as at the end of `compress` in Fig. 2.

Our algorithm uses information extracted from the compiler and our own analysis to infer automatically, for each statement, the PCS before the statement, a sequence of PCS operations applied before the statement, and the PCS after the statement, that is, 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. 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 assertions into the intermediate verification language Viper [40] 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 (unless explicitly assigned to in the program text), 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, implicitly dropped if not required, but never duplicated.

Viper's logic is based on implicit dynamic frames [50], a close relative of separation logic [45], but with the important facility to incorporate heap-dependent expressions in logical assertions (including calls to side-effect-free functions) [51]. 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 [18]); 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, Viper provides two other kinds of 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 value in Rust as a Viper object (of **`Ref`** type); each transitive element of the Rust value (struct fields, tuple elements, box contents, reference targets) is modelled as a field of the Viper object. Furthermore, since with references Rust allows one to take the address of variables, we model references with an additional indirection; the references is modelled as a Viper object with a single field that contains the referenced value.

For example, the parameter `segm` in Fig. 2 is modelled as a Viper **`Ref`** with a field `val: `**`Ref`** for the stored pair. The pair is modelled as an object, with fields `elt0: `**`Ref`**`, elt1: `**`Ref`**; in turn, each of these values is a box, modelled as an object with a single field `box_val: `**`Ref`**. Similarly to pairs, `Point` struct values are 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; bounds are encoded as 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 [44]; predicate *instances* are tracked as 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 non-primitive types, 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:

```
field val_i32 : Int

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

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

```
predicate box_point(self: Ref) {
  acc(self.val_ref) && point(self.val_ref)
}
```

Modelling Place Capabilities As explained in the previous subsection, Rust types prescribe the available capabilities at the entry point to a function, but the PCS may change throughout the function. Using the predicates defined above, we can straightforwardly 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 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, where the Rust capabilities are represented through specifications: we translate the Rust parameter types to a Viper precondition and the result types to a postcondition. Moreover, the Viper embedding of the PCS at each loop head provides a loop invariant that lets Viper verify loops for our core proof automatically.

Many program verifiers, including Viper, prevent indefinite unrolling of recursive predicates by treating predicates isorecursively; 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, where the Rust type checker performs the packing and unpacking PCS operations. Using these statements, as well as method pre and postconditions and loop invariants, Viper constructs the core proof fully automatically, without any user interaction. Recall that full automation is essential to preserve the abstraction provided by Rust and to shield programmers from the complexity of the underlying logic.

Modelling Capability Transfer A method call in Viper checks that the resources required by the precondition are available in the current state (verification fails if they are not) and removes them; conversely, it adds the resources provided by the postcondition of the callee method. This corresponds precisely to the transfer of capabilities in Rust. 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, which provides sound framing.

Resources can also be removed or added explicitly in a Viper program via dedicated **exhale** and **inhale** statements. These allow us, for instance, to encode the move assignment in line 9 of Fig. 2 as **exhale** box_point(segm.val.elt1); **inhale** box_point(end).

3.3 Functional Specifications

The core proof we have constructed so far shows memory safety. This result, by itself, does not go beyond the guarantees provided by the type system. However, the core proof provides the foundation for verifying stronger properties such as functional correctness. In particular, it provides precise aliasing information and framing. Extracting this information from the Rust type system without encoding the full capability information would result in exceedingly complicated assertions. On the other hand, extending our core proof to properties beyond memory safety is surprisingly simple.

We enable checking generic properties such as absence of overflows and absence of panics (e.g. assertion failures) simply, by 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, to check 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 assertions to call side-effect-free Rust functions, similarly to JML's pure methods [30], 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 two sections.

4 Incorporating Borrows

One of the most important and intricate features of Rust's type system is *borrowed references*: 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 borrows; the verification of functional properties will be discussed in the next section. In this paper, we focus on mutable references, including the challenging case of *reborrows*: borrowed references taken from an existing borrow. Extending to shared borrows is future work.

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;

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

Figure 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 `borrow_nth`, which returns a reborrowed reference to a point in the route. Both reborrows expire after the call to `shift_x` on line 49. Functions annotated with `[pure]` are side-effect-free, which can be used in specifications (and may take shared borrows as arguments). The missing ??? specification will be explained in Sec. 5.

that is, the capability for the parameter `r` is transferred to the function, whereas ownership still resides with the caller. Each borrow has a *lifetime*, which is a subset of the program points of the function. At the end of the lifetime, the borrow *expires*, and the capabilities are restored to the original place. Since `r` 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 `borrow_nth` reborrows `r` to transfer the capability to that function. More interestingly, function `borrow_nth` creates a reborrow to a sub-place of the route, namely its nth point, and returns this reborrowed reference to its caller; this reborrow *persists beyond the call in which it is made*. After the call to `borrow_nth`, `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 lifetimes of existing borrows: 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 `borrow_nth` is extended until the further reborrow `p` expires after the call to `shift_x`[1].

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 inadequate for programs with borrows; in particular, it does not explain how, when borrows *expire*, the capabilities (and corresponding permissions in our Viper encoding) can be understood to be 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.

[1]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 newer notion of *non-lexical lifetimes* [1] is more fine-grained; our work supports both.

We represent this information as follows: we assign identities to each borrow operation in a function, as well as every move assignment of a borrowed reference (which we treat as a further reborrow), and every function call which *returns* a reborrowed reference. 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. We obtain this information from the latest borrow checker implementation [56] and additional compiler analyses.

4.2 Encoding Rust Borrows as Resource Assertions

The place capabilities associated with a borrowed reference are (while the borrow is live) encoded just like those for an owning reference (Sec. 3.2); we define a Viper predicate to represent each type of borrowed reference used in the program. Analogously to move assignments, a (re)borrow operation transfers all or some (when borrowing a sub-place) of the capabilities associated with the borrowed-from place; this transfer, however, is only until the (re)borrow expires. We explain how we extend our core proof to reflect this restoration in Sec. 4.3; we consider first how to model reborrows *returned* by functions, such as `borrow_nth` in Fig. 3.

Intuitively, when such a function creates and returns a reborrow then it takes the capability for a place (here, parameter `r`) and *splits it* into two parts: the capability for the reborrow 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 analysis; instead, this remainder is simply unusable *until* the reborrow in question expires. However, to represent this remainder formally in our core proof, we need a suitable formal model for these remainder capabilities, and one which we can define automatically.

Our key insight here is that the separation logic magic-wand connective [42] $-\!\!*$ lets us express *partial* permissions to data structures, such as the `Route` with one `Point` missing. A magic wand assertion $A -\!\!* B$ represents a resource which can be *combined* with the resource A, and $A -\!\!* B$ 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 reborrow, 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.

For functions which return reborrowed references, we now 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, (3) a magic wand $A -\!\!* B$, where A is the same assertion as (1), and B is the Viper embedding of the place capabilities for the *borrow parameters*. For example, for the `borrow_nth` function of Fig. 3, we generate[2]:

```
method borrow_nth(r:Ref, n:Ref) returns (res:Ref)
  requires refRoute(r) && i32(r) &&
    0 <= n.val && n.val < length(r)
  ensures refPoint(res) &&
    res.val.x == old(get_nth_x(r,n)) &&
    (refPoint(res) --* refRoute(r))
```

where `refPoint` and `refRoute` are the predicates generated for structs `Point` and `Route`, resp. The magic wand 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`.

4.3 Automating Proofs with Borrows

Viper supports the magic wand connective, but requires annotations in order to reason about it [49]. 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 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 via function call, the replacement is encoded by *applying* the corresponding magic wand, directed in Viper via an explicit **`apply`** statement in the program.

Consider for example the program point after the call to `shift_x` in function `shift_nth_x` in Fig. 3. At this point, the borrows to `p` and `r` expire. Based on the information we record, we can determine that `p` may be reborrowed from `r`, so we apply the wand `refPoint(res) --* refRoute(r)`, which was returned by function `borrow_nth`.

Creating Reborrows When verifying the definition of a function returning a reborrowed reference (such as `borrow_nth` above), our core proof needs to create the required magic wand, which is done in Viper using a `package` statement.

[2]In fact, 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.

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, and as a side-effect, 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.

We now have the machinery in place to construct the core proof for Rust code that may include mutable borrows and reborrows. We use similar techniques to handle reborrows inside loops 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 verify functional properties of borrows, such as the postcondition of function `shift_nth_x` in Fig. 3.

5 Pledges for Reborrowed References

In Fig. 3 the `shift_nth_x` function performs a shift of a single `x`-coordinate in a `Route`, by obtaining a borrow on the appropriate point (by calling `borrow_nth`), and then using this borrowed reference to mutate the point (via a call to `shift_x`). While this is a simple example, it illustrates the challenges of reasoning modularly about code mixing mutability and borrows. As a verification goal, we have written a strong functional specification for `shift_nth_x`, expressing that the resulting `Route r` will be the original except for the single shifted coordinate[3]. However, we cannot prove this specification using the current specification of `borrow_nth`. Considering the call to this function on line 48, it becomes clear that the postcondition of `borrow_nth` is not yet useful: for this call, it expresses a relationship between the coordinate of the returned reborrow `p` and an original point in the `Route r` passed, but does not guarantee how *changes* to `p` will affect the state of the `Route`, nor whether and how the remainder of the `Route` has been changed by the call to `borrow_nth` (in this case, it is unchanged).

Expressing such a specification is challenging. We want a formal guarantee about a *future* point in the program's execution: we want to describe how the borrowed-from `Route r` *will* look after the reborrow `result` (the value returned from the function) expires. But from the point of view of the function `borrow_nth`, we cannot yet know this information fully; the reborrow that the function returns can (and in our example will, on line 49) be used *in the meantime* to modify the borrowed-from `Route`. A different caller of this function might make different modifications, and a proof about its behaviour would equally rely on knowing how these correspondingly affect the borrowed-from `Route` passed as parameter. Note that directly expressing a postcondition for `borrow_nth` in terms of `r` *at the this time* is not a suitable solution for three reasons: (1) `r` is not (in general) accessible immediately after a call to this function (so such a specification would violate Rust's typing rules), (2) in general, memory that is still borrowed-from could be currently accessible via other threads; such a specification would have an undefined semantics, and (3) such a specification would not provide the missing connection between how *subsequent* mutations via still-active reborrows entail corresponding changes to the borrowed-from memory.

Pledges To address this problem, we present a novel specification construct which can directly express how, when a reborrow expires, the reborrowed memory *will be guaranteed* to look, *parametrically with* the state of the reborrow *by the time* it expires. We call this construct a *pledge*. A pledge is written using the syntax `afterExpiry<x>(b)`, where x is the reborrow the pledge concerns, and b is the property which will be true *after* the expiry of the reborrow. To enable a pledge to describe a property parametric with the final state of the *reborrowed memory*, inside b the additional construct `beforeExpiry<x>(e)` can be used to denote the value of an expression *immediately before* the reborrow expires. In both cases, the `<x>` parameter can be omitted where unambiguous (e.g. all returned reborrows expire at the same time).

Fig. 4 illustrates how our new feature enables a postcondition for `borrow_nth` from Fig. 3 which is both sufficiently strong to prove the original code correct, and generic in the sense that other callers which might make different usage of the reborrowed reference can use it equally. The pledge (which makes up the second postcondition), expresses the guarantee that *once* the returned reborrow expires: (1) the length of the borrowed-from `Route r` will be unchanged, (2) the `x`-coordinate of `r`'s n^{th} `Point` will be equal to that of the returned `Point` reborrow *when the reborrow expires*, and (3) all other `x`-coordinate of the `Route` will be unchanged. Note that properties (1) and (3) are guaranteed by the fact that reborrows to the *rest* of `r`'s memory are *not* returned.

The pledge is conceptually analogous to a postcondition for the *reborrow* (*cf.* the postcondition of `shift_nth_x` in Fig. 3); the parameterisation with respect to the borrowed data allows the pledge to provide a guarantee in terms of what happens between now (the reborrow being created) and then (the borrow expiring). This makes the pledges a particularly good match for getter methods in an interface, but they could equally well apply to code which combines side-effects before creating the reborrowed reference with the potential for side-effects via the reborrow itself; the pledges allow one to formally specify the guaranteed overall effect.

The encoding of our verification technique to Viper extends naturally to handling pledges; the pledge is translated as an additional conjunct on the right-hand-side of the generated magic wand (recall that this indeed represents the state as it *will* look once the reborrow expires, while the left-hand-side of the magic wand models the capabilities returned

[3]We omit information about the unchanged `y`-coordinates, for the exposition.

```
#[requires="0 <= n && n < length(r)"]
#[ensures="result.x == old(get_nth_x(r, n))"]
#[ensures="afterExpiry<result>(
  length(r) == old(length(r)) && get_nth_x(r, n)
    == before_expiry<result>(result.x) &&
  forall i: i32 ::
    (0<=i && i<length(r) && i != n) ==>
     get_nth_x(r, i) == old(get_nth_x(r, i))) "]
fn borrow_nth(r:&mut Route, n: i32) ->
 &mut Point {
...
```

Figure 4. An example of our pledge specification feature.

from the reborrow). When such a magic wand is packaged (*cf.* Sec. 4.3), we effectively force Viper to prove that the claimed pledge will indeed hold for *any* future state of the reborrowed memory, which is necessary to soundly account for any possible usage of the still-active reborrowed reference. Viper provides expressions **old**[lhs](e) which, when used on the right-hand-side of a magic wand, evaluate to *e*'s value in the state corresponding to the wand's left-hand-side; this enables us to express our before_expiry construct by generating a corresponding Viper-level **old**[lhs](e) expression.

6 Implementation and Evaluation

We have implemented our work as a plugin for the Rust compiler, and evaluated it using crates from a wide variety of popular repositories (crates are analogous to Java packages).

6.1 Implementation

We implemented a tool called Prusti [2] as a Rust compiler plugin, usable with Cargo, the official package manager for Rust. This setup closely resembles existing tools such as Clippy [12], the official linter for Rust; working with Prusti provides a similar experience to existing tools used by Rust developers. 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 reported using the Rust compiler's error reporting mechanisms. In addition to proving user specifications, Prusti can be optionally configured to check absence of panics and overflows.

Our current tool works on a subset of Rust; in addition to the more-fundamental restrictions discussed (e.g. verifying unsafe code), we have not yet implemented support for struct fields storing borrowed references, or loops with **break**, **continue** or **return**s in the body; these restrictions will be lifted in the future (and can typically be worked around by rewriting the program). Since libraries such as **Vec**

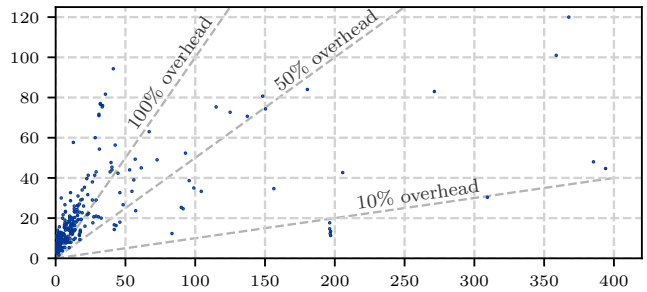

Figure 5. Overall Prusti overhead (vertical axis, in seconds) compared to baseline compilation duration (horizontal axis, in seconds). Each dot represents a crate.

(vectors) are used ubiquitously in Rust, our tool supports a *#[trusted]* annotation, allowing us to equip functions with contracts used by callers but *not* checked against the function's implementation; this allows us to wrap and provide specifications for commonly-used library types.

6.2 Evaluation

We evaluate our work in three ways: (1) we evaluate the construction of our core proofs: we scraped the top 500 Rust crates for all functions within our supported subset, and had our tool construct core proofs for these functions without specifications; (2) we enabled checking for absence of overflows (in examples which make runtime checks), to see how many our tool can prove redundant (with no specifications); (3) we took existing implementations of well-known algorithms and attached specifications of these to verify absence of panics as well as richer functional correctness properties. 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, 32 GB of RAM and an SSD disk.

(1) Core Proofs To test the automation and overhead of our core proof construction, we took the 500 most popular Rust community crates [13], and applied two 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 56367 functions (top-level, **impl** and trait functions) with a simple syntactic check for unsupported language features; this left us with 2111 candidate functions to evaluate our work on. We re-ran the compiler with Prusti on the unmodified source code of these functions, causing core proofs to be generated and verified. The verification of all 2111 functions succeeded as expected, without any need for manual intervention. This shows that we generate sufficient annotations to explain the core capability proof to Viper. These annotations are non-trivial: summing over these functions, 91123 lines of

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

Viper code were generated, of which 14242 are **fold**, **unfold**, **package** or **apply** statements to automate the proofs.

We measured how much wall-clock time compiling (and verifying) with Prusti takes, compared to the baseline compiler, reporting average results (across three runs, between which the compiler cache was cleared) in Fig. 5. We observe that on quickly-compiling crates the overhead of running Prusti is significant, although still typically below 40 seconds. This relatively high overhead is due to our currently-coarse interaction with Cargo; we run verification on some compiled *dependencies* of a crate, as well as the crate itself (starting a JVM process each time for the verifier). We can clearly reduce this overhead in the future. For longer-running crates, our overheads do not rapidly increase; since our verification is modular, verification time can be expected to scale linearly with the number of functions.

(2) Overflow Freedom Of the functions verified in part (1), we identified those containing code which potentially raised panics due to overflows. There are 71 such functions; we reran Prusti on these, enabling checks for panics and overflows (again, without specifications). Interestingly, 21 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 these can never fail, one could use this to justify the elimination of the runtime checks, improving performance without compromising safety.

For each of the remaining 50 functions Prusti reported a verification error: potential overflows in 48 cases, 1 potential division by zero and 1 potentially failing **assert!** statement. Manual inspection showed these to rely on implicit assumptions on argument ranges; our tool makes it possible for developers to easily make these assumptions explicit (as preconditions), and verified at each call site.

(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 [14], and from Matsakis' blogs on Rust's language design [36, 37]. Using the same filtering as for part (1), we identified examples from Rosetta Code containing at least one supported functions, manually discarding the examples that also make use of functions with unsupported features: IO operations, closures and iterators.

In order to handle examples using standard library types (whose implementations we cannot verify), 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. 6 gives an overview of the verified examples. 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 first (so far unoptimised) tool and translation. The selection sort example takes just over this time, Heapsort takes 34 seconds and the slowest example (Knight's tour) takes 98 seconds (this file contains one monolithic function). In all cases, standard deviations were around 1 second.

Example	LOC	#Fns	No Panic	No Overflow	Time (s) All	Time (s) Viper
100 doors	20	2	✓	✓	11.4	7.5
Ackermann Func.	20	3	✓		7.4	4.2
Binary Search	16	1	✓	✓	27.6	20.5
Fibonacci Seq.	46	6	✓		9.4	5.5
Heapsort	39	3	✓		33.8	27.1
Knight's tour	89	6	✓		97.9	72.2
Knuth Shuffle	16	2	✓		9.0	5.3
Langton's Ant	58	4	✓	✓	17.4	10.6
Towers of Hanoi	10	2	✓	✓	5.8	3.3
Selection Sort	20	2	✓	✓	29.6	20.3
Message	13	1	✓	–	7.1	3.8
Borrow First	7	1	✓	✓	7.0	3.7

Figure 6. An overview over the examples verified in the third part of the evaluation. The column "LOC" indicates the number of lines in the unmodified example; "#Fns" the number of verified functions; "No Overflow" whether we verified absence of overflows ("–" means that the example contains no arithmetic operations); "All Time" indicates the time in seconds required to compile and verify the example; "Viper Time" is just the time needed by the Viper symbolic execution back-end verifier to verify the encoding. The first ten examples are taken from the Rosetta Code website [14]; the last two from Niko Matsakis' blogposts about non-lexical lifetimes in Rust [36, 37].

For all examples we verified absence of panics, which in most examples required proving that vector accesses are within bounds. The most interesting extra specification 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. 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), and equivalence of functional and iterative code (Fibonacci and Ackermann).

We also verified two examples from Matsakis's blog [36, 37], designed illustrate difficult borrowing patterns that will be accepted as valid Rust programs in the near future. Both examples can already be verified by our tool (using the corresponding new borrow checker implementation).

Overall, while we expect to improve the runtimes involved in our verification process, we believe that these results show that our techniques have significant potential for a wide variety of applications, both in validating standard properties such as panic freedom, and for identifying incorrect functional behaviour using Rust-level specifications. Our tool's tight integration with the Rust compiler and surrounding infrastructure, and ability to annotate source code directly, make these examples simple to write and understand: we provide the files as auxiliary material.

7 Related Work

Capability-Based Type Systems Many other type systems can also be understood to associate *capabilities* with reference types [9]. Some extend pre-existing languages (e.g. Sing♯ [5], C♯ [19] and Scala actors [20]); more recently, several programming languages have built these in (e.g. Pony [10], AEminium [53], and Rust itself [38]). 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 [10]. 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 [48] equip types with logical qualifiers prescribing value properties; their extension to *Alias Refinement Types* [3] applies to mutable heap-based data structures. Type checking is decidable, and loop invariants can be inferred. Unlike our work (*cf.* Sec. 4, Sec. 5), there is no support for references (reborrows) which persist beyond the function calls or loops they are created in (common in Rust, *cf.* Fig. 3); this is future work in [3].

SYMPLAR [7] 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 [34], using a type system similar to Rust. In both systems, however, returning reborrowed references is not supported.

Rust Verification Tools CRUST [58], a recent adaptation of SMACK [4], and Lindner *et al.* [33] provide *bounded* verification tools for Rust (including unsafe code); the latter two 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* verification, and support richer functional specifications via `old` expressions and pledges.

Ullrich [59] encodes safe Rust programs into *functional programs*, to be interactively verified in Lean [15]. Reborrows are supported via *lenses* [17]. Recent work at Galois similarly reduces reasoning about a safe subset of Rust to proofs about functional programs in Saw [16]. In contrast to these works, our techniques do not require manual construction of proofs or verifier directives; in addition, our underlying separation logic formalism will provide a suitable (imperative-style) model for extension to forms of unsafe code in 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 themselves and the starting points for our work.

Rust Semantics and Formalisations A number of formalisms for subsets of Rust have been designed, focusing on type soundness results [27, 46, 60, 61]. It would be interesting to compare these formal models with the PCS/borrow summaries that our work produces from the compiler.

Rustbelt [25] 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 techniques enable 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 shared 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.

8 Conclusions and Future Work

We presented a new specification and verification technique for Rust, leveraging the guarantees providing 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, while being shielded from the complexities of the underlying program logic. 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, and our evaluation shows that we can reliably automatically construct core proofs for real-world Rust code, and detect previously-unknown bugs via failed attempts to prove functional specifications.

Our main goal for future work is to extend the subset of Rust supported by our techniques and tool. Handling Rust's shared borrows is an important next step, which we can at least partly tackle using fractional permissions [8] in our core proofs. We plan to tackle verification for other prevalent Rust features, such as iterators and closures; these extensions together will broaden the applicability of our techniques to a much wider range of real-world code. We also plan to address support for certain classes of *unsafe* code; intriguingly, it has also been observed that even being able to specify and prove invariants in *safe* code may go a long way towards eliminating problems manifesting in unsafe code which relies on such invariants [24, 35].

References

[1] Niko Matsakis (and contributors). 2017. Non-Lexical Lifetimes RFC. https://github.com/rust-lang/rfcs/blob/master/text/2094-nll.md

[2] Anonymous Authors. 2018. A Virtual Machine With Our Implementation. sha256sum: c420f6f3446b4adf08d79cd0e9bbb8ffbf8ed09961e6b508400d2fa15ef5a5c9. (link removed for double blind review).

[3] 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

[4] 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.

[5] 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

[6] 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.

[7] 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

[8] John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*, Vol. 2694. Springer, 55–72.

[9] 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.

[10] 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

[11] 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.

[12] Clippy contributors. 2018. Clippy. https://github.com/rust-lang-nursery/rust-clippy

[13] Crates.io contributors. 2018. The Rust community's crate registry. https://crates.io

[14] Rosetta Code contributors. 2018. Rosetta Code. https://rosettacode.org/wiki/Category:Rust

[15] 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*.

[16] 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.

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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.

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

[24] Ralf Jung. 2016. The Scope of Unsafe. https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. In *Proceedings of the 6th ACM SIGPLAN Workshop on Higher-Order Programming. Oxford, United Kingdom.*

[26] 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.

[27] 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

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

[29] 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

[30] 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/.

[31] 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

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

[33] M. Lindner, J. Aparicius, and P. 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

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

[35] Niko Matsakis. 2017. (Personal Communication).

[36] Nicholas D. Matsakis. 2018. MIR-based borrow check (NLL) status update. http://smallcultfollowing.com/babysteps/blog/2018/06/15/mir-based-borrow-check-nll-status-update

[37] Nicholas D. Matsakis. 2018. MIR-based borrowck is almost here. http://smallcultfollowing.com/babysteps/blog/2018/10/31/mir-based-borrowck-is-almost-here/

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

[39] B. Meyer. 1992. Design by Contract. In *Advances in object-oriented software engineering*, D. Mandrioli and B. Meyer (Eds.). Prentice Hall, 1–50.

[40] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*, B. Jobstmann and K. R. M. Leino (Eds.), Vol. 9583. Springer-Verlag, 41–62.

[41] 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.

[42] 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

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

[44] M. Parkinson and G. Bierman. 2005. Separation logic and abstraction. In *POPL*, J. Palsberg and M. Abadi (Eds.). ACM, 247–258.

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

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

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

[48] 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

[49] M. Schwerhoff and A. 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.

[50] J. Smans, B. Jacobs, and F. Piessens. 2009. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP (LNCS)*, Vol. 5653. Springer, 148–172.

[51] 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.

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

[53] 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

[54] Coq Development Team. 2014. The Coq Proof Assistant Reference Manual. http://coq.inria.fr.

[55] The Rust Team. [n. d.]. Rust: The Reference — Place Expressions and Value Expressions. https://doc.rust-lang.org/reference/expressions.html#place-expressions-and-value-expressions

[56] The Rust Team. 2018. The Polonius Reference Implementation for the Rust Borrow-Checker. https://github.com/rust-lang-nursery/polonius

[57] The Rust Team. 2018. The Rustonomicon : Working with Unsafe. https://doc.rust-lang.org/nomicon/working-with-unsafe.html

[58] 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.

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

[60] 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

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