
# RefinedRust: A Type System for High-Assurance Verification of Rust Programs

LENNARD GÄHER, MPI-SWS, Germany
MICHAEL SAMMLER, MPI-SWS, Germany
RALF JUNG, ETH Zurich, Switzerland
ROBBERT KREBBERS, Radboud University Nijmegen, Netherlands
DEREK DREYER, MPI-SWS, Germany

Rust is a modern systems programming language whose ownership-based type system statically guarantees memory safety, making it particularly well-suited to the domain of safety-critical systems. In recent years, a wellspring of automated deductive verification tools have emerged for establishing functional correctness of Rust code. However, none of the previous tools produce *foundational* proofs (machine-checkable in a general-purpose proof assistant), and all of them are restricted to the *safe* fragment of Rust. This is a problem because the vast majority of Rust programs make use of *unsafe* code at critical points, such as in the implementation of widely-used APIs. We propose *RefinedRust*, a refinement type system—proven sound in the Coq proof assistant—with the goal of establishing *foundational* semi-automated functional correctness verification of both safe and unsafe Rust code. We have developed a prototype verification tool implementing RefinedRust. Our tool translates Rust code (with user annotations) into a model of Rust embedded in Coq, and then checks its adherence to the RefinedRust type system using separation logic automation in Coq. All proofs generated by RefinedRust are checked by the Coq proof assistant, so the automation and type system do not have to be trusted. We evaluate the effectiveness of RefinedRust by verifying a variant of Rust's `Vec` implementation that involves intricate reasoning about unsafe pointer-manipulating code.

CCS Concepts: • **Theory of computation** → **Program verification**; **Separation logic**.

Additional Key Words and Phrases: separation logic, program verification, Rust, Iris

**ACM Reference Format:**
Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 192 (June 2024), 25 pages. https://doi.org/10.1145/3656422

## 1 INTRODUCTION

Rust [47] is a modern systems programming language that is seeing increasingly widespread adoption in industry as an essential tool for building more trustworthy systems code [2, 45]. One of Rust's key selling points is that its core type system guarantees *memory safety*, thus ruling out common errors made by programmers in legacy systems programming languages like C and C++, without compromising on performance. Indeed, Rust's memory safety guarantees are a primary factor driving its adoption in safety-critical systems like the Linux kernel [50].

Authors' addresses: Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Ralf Jung, ETH Zurich, Zürich, Switzerland, research@ralfj.de; Robbert Krebbers, Radboud University Nijmegen, Nijmegen, Netherlands, mail@robbertkrebbers.nl; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.


ACM 2475-1421/2024/6-ART192
https://doi.org/10.1145/3656422

Of course, for safety-critical programs, it is not ultimately sufficient that they are memory-safe—we also want to establish that they do what they are supposed to do, *i.e.*, that they satisfy *functional correctness* properties. Toward that end, there has emerged a wellspring of exciting research in recent years, leading to a range of new verification tools, such as Prusti [3], Creusot [9], Flux [28], and Aeneas [12]. These tools have made impressive strides forward, particularly in leveraging the expressive power of the Rust type system to simplify the task of verifying Rust programs.

However, all the aforementioned verification tools share two key limitations. One is that, like most practical verification tools, they are standalone software artifacts, the implementations of which are increasingly complex and thus add significantly to the trusted computing base (TCB) of any verification conducted with them. The other limitation pertains to the handling of *unsafe* code. Although Rust is renowned for its safety guarantees, its type system is sometimes overly restrictive: there are certain systems programming idioms which cannot be implemented in safe Rust. As a result, many *observably safe* Rust APIs are implemented internally with sparing use of *unsafe* features of the language (such as raw pointer manipulations or unchecked type casts, which may result in undefined behavior). Yet none of the previous verification tools for Rust (see §7 for a comparison to GillianRust [55], which was developed concurrently with RefinedRust) support the verification of Rust APIs implemented with unsafe code.

Ideally, we would like to develop technology for verifying Rust programs that avoids both of these limitations—*i.e.*, that handles unsafe code, and that lowers the TCB by producing *foundational* proofs in a proof assistant such as Coq—while retaining support for automated verification.

To that end, we present **RefinedRust**, a new approach to the foundational verification of Rust programs, based on *refined ownership types* [41]. RefinedRust is the first approach to Rust verification that simultaneously (1) handles real (surface) Rust code, (2) provides support for proof automation, *both for safe and unsafe Rust code*, and (3) outputs machine-checkable proofs for all verified code. We have implemented a prototype of RefinedRust in Coq. Its automation support is fairly basic compared to that of previous non-foundational Rust verification tools, but no previous tool (foundational or otherwise) supports any automation for verifying unsafe Rust code, and thus RefinedRust makes an important first step.

As the name suggests, the starting point for RefinedRust is Sammler *et al.*'s earlier work on RefinedC [41], a system for verifying functional correctness of *C programs* that is both foundational and semi-automated. RefinedC achieves this goal by developing an extension of C's type system with *refinement* and *ownership* types, which enable it to express rich functional specifications on the behavior of C code. RefinedC defines a semantic model of its refinement types in the separation logic Iris [21, 19, 25, 20, 26, 44], so that the soundness of RefinedC type checking is established foundationally in Coq. Moreover, RefinedC's typing rules are expressed in a fragment of Iris called Lithium [40], which is carefully designed to admit efficient proof search without backtracking; as a result, RefinedC type checking can be performed with a (relatively) high degree of *automation*.

At a high level, the idea behind RefinedRust is to take RefinedC's approach of refined ownership types and figure out how to make it work for Rust. The most obvious challenge in doing so is developing useful refinement types (and typing rules) to automate reasoning about Rust's most distinctive feature: its reference types, along with their attendant notions of *lifetimes* and *borrowing*. Towards that end, we take inspiration from RustBelt [18], leveraging its *lifetime logic*.

But of course the devil is in the details. In developing RefinedRust, we had to overcome a number of technical challenges, related to: (1) bridging the gap between Rust and the RustBelt model, and (2) adapting RefinedC's refinement type system to handle Rust types.

**Challenge #1: Bridging the gap between Rust and RustBelt.** As explained above, RefinedRust achieves foundational verification by giving a semantic model of Rust types as predicates in the

Iris separation logic [48]. This semantic model is inspired closely by that of RustBelt, but RustBelt employs an idealized formalization of Rust called $\lambda_{\text{Rust}}$. In order to account for real Rust code, we had to overcome two key gaps between RustBelt/$\lambda_{\text{Rust}}$ and Rust.

First of all, RustBelt makes no attempt to capture the notion of "places" in the Rust language (also known as "lvalues" in C). Places occur on the left-hand side of assignments and as the operands of the "address of" operator `&`; for instance, in `&x.f`, the expression `x.f` denotes the place in memory where the `f` field of variable `x` is stored. In RustBelt, to port a Rust program to $\lambda_{\text{Rust}}$, one must replace uses of general places by a few simple cases that the RustBelt type system can handle. This requires manual effort and fails to properly reflect the structure of the Rust source code.

Secondly, $\lambda_{\text{Rust}}$ does not accurately reflect all aspects of Rust code. For instance, integers in $\lambda_{\text{Rust}}$ are unbounded (as opposed to Rust's real semantics with integer overflows), and $\lambda_{\text{Rust}}$ does not accurately reflect how data is represented in memory, especially for compound types like structs.

RefinedRust lifts both of these limitations. As a more realistic operational semantics suitable for functional verification of unsafe code, RefinedRust introduces **Radium** (based on RefinedC's Caesium model for C [41]). Hence, programs verified in RefinedRust are proven to correctly deal with intricacies such as integer overflows. To ensure correctness independent of the concrete data layout, RefinedRust parameterizes its verification by an arbitrary "layout algorithm" (§4). And to properly account for the role places play in Rust, RefinedRust's type system introduces *place types*. This aligns RefinedRust sufficiently well with the Rust type checker that we can automatically translate Rust code into Radium and type check the result with RefinedRust.

**Challenge #2: Extending RefinedC with refinement types for Rust's mutable references.** The basic structure of RefinedRust is modeled after that of RefinedC: it layers a refinement type system on top of Rust's type system, and then expresses its refinement type checking rules in the Lithium fragment of Iris to make it amenable to proof automation. However, Rust is a very different language from C, so "porting RefinedC to Rust" is far from straightforward. In particular, C only has a weak type system that describes the layout of values in memory, but does not give strong guarantees or provide many mechanisms for abstraction. RefinedC therefore introduces its own type system with a Rust-inspired notion of *exclusive ownership*. However, the Rust type system goes well beyond exclusive ownership, using *borrowing* to grant temporary access to data without full ownership transfer. Borrowing in Rust is expressed via reference types: *shared references* `&T` for immutable borrowing, and *mutable references* `&mut T` for mutable borrowing.

Extending the RefinedC type system to handle Rust's shared references is fairly straightforward, but mutable references constitute a major challenge. To explain why, we have to briefly consider how RefinedC represents a variable with a known value: $42 \mathbin{@} \mathsf{int}_{\mathsf{i32}}$ is the (singleton) type of an integer with value 42. Similarly, $\&\mathsf{own}(42 \mathbin{@} \mathsf{int}_{\mathsf{i32}})$ is the type of an exclusively owned pointer that points to an integer with value 42. Note how the owned pointer type entirely wraps the integer type, *including its value*. If the program changes the value stored behind that pointer to 57, RefinedC uses a "strong update" (*i.e.*, type-changing update) to enable the pointer type to be changed accordingly to $\&\mathsf{own}(57 \mathbin{@} \mathsf{int}_{\mathsf{i32}})$. Such a strong update is sound precisely because the pointer is *exclusively owned*: there is no risk that any other part of the program could have a different view on this data that would be in conflict with a strong update.

In contrast, a mutable reference in Rust is not exclusively owned—rather, it is borrowed from somewhere, and once the borrow expires, the original owner will want to use that data again *at its original type*. This problem is an instance of the common pattern that when a reference type allows for shared state (in this case, state that is shared between the borrower and the original owner), the type must be invariant. As a result, Rust's mutable references do not allow strong updates, and the RefinedC strategy for precisely tracking the values stored in them no longer works.

In RefinedRust, we therefore keep the value "outside" of the mutable reference: a mutable reference that points to an integer with value 42 would, roughly, have type $42 \mathbin{@} \&_{\text{mut}} \mathsf{int}_{\mathsf{i32}}$. This lets us change the value without performing a strong update on the mutable reference type. However, this does not solve the issue that eventually, the lifetime of the reference will expire, and then the information about the new value stored behind that reference has to be propagated back to its original owner. To this end, RefinedRust introduces *borrow names*, which are inspired by RustHorn's *prophecy variables* [32]: the full type of a mutable reference takes the shape $(42, \gamma) \mathbin{@} \&_{\text{mut}} \mathsf{int}_{\mathsf{i32}}$ where 42 is the current value, and $\gamma$ is a borrow name that lets the original owner incorporate changes to that value into their own proof (§2.2). This change represents a fundamental departure from RefinedC in terms of how type constructors interact with refinements, requiring a redesign of large parts of the type system and a non-trivial extension to RustBelt's lifetime logic.

**Contributions.** We present **RefinedRust**, a new foundational approach for verifying functional correctness of safe and unsafe Rust programs. We make the following conceptual contributions:

- We show how to handle Rust's mutable references in a refinement type system, using the novel mechanism of *borrow names* to link the value of a mutable reference with the value of the borrowed place (§2.2, §2.3).
- To support the borrowing patterns that appear when verifying actual Rust code, we build a place type system for RefinedRust including novel types that enable Rust-specific reasoning about borrowed places and types with invariants (§3.2, §5.2, §5.3). For our soundness proof of the type system, we have extended RustBelt's lifetime logic with a new kind of borrows (described in the supplementary material [11]).
- To verify Rust's polymorphic functions and to support layout-generic verification, we develop a semantics which parameterizes the code with type parameters and a layout algorithm. The verification then happens generically in the layout algorithm and type parameters (§4.1).

Additionally, we provide an implementation of RefinedRust with the following components:

- The RefinedRust type system, a type system for Rust that combines refined ownership types with a semantic model of Rust types inspired by RustBelt (§5).
- Radium, a formalization of Rust, based on RefinedC's Caesium operational semantics (§4).
- A type checker for RefinedRust based on RefinedC's Lithium proof engine in Coq, and a frontend that translates Rust code to Radium in Coq by leveraging the Rust compiler (§6).
- We evaluate RefinedRust on a version of the Rust standard library's `Vec` vector implementation (§3, §6). The code has been simplified for engineering reasons, but it still captures many of the intricate challenges of working with pointer-manipulating unsafe Rust code.

The RefinedRust type system and the RefinedRust type checker are mechanized in Coq using Iris. §5.4 describes the high-level soundness result—the technical details, including RefinedRust's extensions to RustBelt's lifetime logic, can be found in the supplementary material [11].

**Non-goals and limitations.** The RefinedRust prototype produces proofs directly in Coq, which means that its implementation does not need to be trusted. This also limits the approaches that we can use for automation (*e.g.*, SMT solvers as trusted oracles) compared to other verification tools for Rust, such as Creusot [9] and Prusti [3]. While RefinedRust closes some gaps between RustBelt and real Rust, there are still aspects of the Rust semantics that we do not yet account for, such as some details of Rust's layouting of structs and enums (*e.g.*, the niche optimizations [4]), some of Rust's validity invariants (*e.g.*, that booleans values are always 0 or 1), pointer-integer casts, and the aliasing model [17] (which remains a topic of active research). A complete and precise specification of the operational semantics of Rust does not exist yet. RefinedRust does not support some more advanced features of Rust such as concurrency, recursive types, traits, closures, and unsized types.

```
1 #[rr::params("x" : "Z")]
2 #[rr::args("#x")]
3 #[rr::requires("x + 42 ∈ i32")]
4 #[rr::returns("#(x + 42)")]
5 fn box_add_42(mut r: Box<i32>) -> Box<i32>
6 { *r += 42; r }
```

Fig. 1. Function that adds 42 to an integer stored in a box.

```
1 #[rr::params("x" : "Z", "γ" : "gname")]
2 #[rr::args("(#x, γ)")]
3 #[rr::requires("x + 42 ∈ i32")]
4 #[rr::ensures(#iris "Res γ (x + 42)")]
5 fn mut_ref_add_42(r: &mut i32)
6 { *r += 42 }
```

Fig. 2. Function that adds 42 to an integer stored in a mutable reference.

## 2 AN INTRODUCTION TO REFINEDRUST

We outline the basic principles of RefinedRust using our prototype implementation (described in §6). Then we introduce our notion of refined ownership types (§2.1), and their combination with Rust's mutable references (§2.2 and §2.3). Finally, we show our handling of unsafe functions (§2.4).

### 2.1 Refinement Types

Inspired by RefinedC, RefinedRust enables functional correctness verification through *refinement types*. Let us explain refinement types in RefinedRust by considering the function `box_add_42`, shown in Figure 1. It takes an argument `r` of type `Box<i32>`, where `Box<T>` is Rust's type for an owned pointer to a value of type `T`. The function adds 42 to the value stored in `r`, and then returns `r` to the caller. The type `Box<T>` has full ownership of the memory the pointer refers to, which means there cannot be other pointers to the memory. It is thus necessary that `r` is returned to give ownership of the memory back to the caller, otherwise Rust would drop the box and free the memory.

In RefinedRust's type system, all types come with a notion of *mathematical values* that they are *refined by*. Consider the specification for `box_add_42` in Figure 1. Here, refinement type information is given via Rust attributes `#[rr::...]`. First, the `params` attribute introduces a specification variable `x` of the mathematical (unbounded) integer type `Z`. The `args` attribute then links this variable to the function argument `r` of Rust type `Box<i32>`, by stating that `r` is refined by the mathematical integer `x`. (The injection `#` is discussed in §2.3.) The `requires` attribute on line 3 specifies the precondition that `x + 42`—the result of the addition performed by the function—must be in the value range representable by the type `i32`. The precondition ensures that the addition does not overflow, which would trigger a *panic* and abort program execution. The precondition is necessary because RefinedRust also verifies that no panics occur, similar to other Rust verification tools [3, 9]. Finally, the `returns` attribute specifies the mathematical value `x + 42` of the box returned by `box_add_42`.

### 2.2 Mutable References

The function `mut_ref_add_42` in Figure 2 is a more idiomatic version of the previous example. It uses Rust's mutable reference type `&mut T` instead of a `Box<T>` to avoid returning the box. Like `Box<T>`, a mutable reference `&mut T` asserts exclusive ownership of its memory, and thus allows mutating it (*e.g.*, by adding 42). However, this exclusive ownership is limited in time: a mutable reference has a *lifetime*, and only *borrows* the referenced memory for this lifetime. Once the function returns, the lifetime is over and the caller regains ownership of the memory. To illustrate this point, consider:

```
let mut z = 1; mut_ref_add_42(&mut z); assert!(z == 43);
```

The use of `z` in the `assert!` implicitly ends the lifetime of the reference passed to `mut_ref_add_42`. Proving that the `assert!` succeeds requires knowing that `z` is indeed incremented by `42`. However, there is no *explicit* flow of values from `mut_ref_add_42` to the `assert!` since `mut_ref_add_42` returns a unit value. Unlike the previous example, we thus cannot use the `returns` attribute to specify the increment of `z`, and need another way to specify the side-effect of incrementing `z`.

To reason about mutable references, RefinedRust uses the notion of *borrow names* (inspired by RustHorn's *prophecy variables* [32]). The mathematical value of a mutable reference `&mut T` is a pair `(x,γ)`, where `x` is the current mathematical value of type `T` and the borrow name $\gamma$ is used to communicate the final value of the reference. In our example, `&mut z` will create a reference with mathematical value `(1,γ)` for some fresh $\gamma$, and the value of `z` changes from `1` to $*\gamma$. In other words, the value "moves" from `z` to the new reference, and $\gamma$ ties the reference to its referent. This reference is then passed to `mut_ref_add_42` (instantiating `x` with 1). The `ensures` clause (*i.e.*, postcondition) of that function is a *resolution* `Res γ (x + 42)`, which states that the final value of the reference with borrow name $\gamma$ is `x + 42` (the `#iris` annotation specifies that the postcondition is a separation logic assertion). This resolution lets RefinedRust automatically resolve the mathematical value of `z` from $*\gamma$ to 43. Thus, it can prove that the `assert!` succeeds.

### 2.3 Reborrows

Now that we have seen the basics of refinement types and borrows in RefinedRust, let us turn to a more advanced use case of mutable references: the method `get_mut` for creating a mutable reference to an element of a Rust vector. In this section, we use `get_mut` to explain the concept of *reborrowing*. In §2.4, we will verify the implementation of `get_mut` based on the unsafe helper function `get_unchecked_mut`, which we in turn verify in §3. The signature of `get_mut` is:

```
fn Vec::get_mut<'a>(&'a mut self, idx: usize) -> Option<&'a mut T>
```

It takes a mutable reference to the vector **`self`** as well as an index `idx`. It checks if `idx` is within bounds of the vector, and if so, returns a reference to that element of the vector (else it returns `None`). The returned reference can be used by the caller to modify this element of the vector, and Rust's borrow checker makes sure that the vector is inaccessible as long as the reference is in use. This is especially clear when considering the lifetimes of the references in the type signature: the vector gets a reference of lifetime `'a`, and the returned reference to the element has the same lifetime `'a`. This means that, as long as the returned reference is in use, the reference passed as argument is also in use. As such, `get_mut` is an example of a *reborrow* function common in Rust, taking a reference as an argument and then providing a "view" into that reference in the return value.

Figure 3 shows a simple client of `get_mut`. It creates a vector `x` containing 100, 200, and 300, and uses `x.get_mut(1)` to get a mutable reference `xr` to the element at index 1. The function `unwrap` takes an `Option<T>` and returns `t` if the input is `Some(t)`, and panics if it is `None`. Since 1 is within range of the vector `x`, such a panic cannot happen. The first `assert!` checks that `xr` indeed refers to 200. After updating the value of `xr` to 42, the second `assert!` checks that the write to `rx` updated the vector `x` as expected. Importantly, in the second `assert!`, the vector is accessed through `x` again, which means that the lifetime `'a` of the reference returned by `get_mut` ends and `rx` cannot be used anymore.

Let us look at the specification for `get_mut` in Figure 3 to see how it enables the verification of the assertions in `get_mut_client`. The first argument **`self`** has the Rust type `&mut Vec<T>`, so it is refined by a list `xs` of (borrowable) mathematical values for `T`,[1] together with a borrow name $\gamma$ for the mutable reference. The second argument `idx` is refined by a mathematical integer `i : Z`. The remaining clauses specify the postcondition: `exists` declares a mathematical variable that is returned by the function (akin to an existential quantifier in the postcondition). The `returns` clause specifies the real return value: if the index `i` is within bounds of the vector, `get_mut` returns `Some` containing a reference to the `i`-th element of `xs` with the fresh borrow name `γi`, otherwise it returns `None`. Here, the syntax `xs !!! i` indicates list indexing.

The most interesting part of this specification is the `ensures` clause: if `i` is out of bounds, the caller receives `Res γ xs`, stating that the vector is unchanged. (Recall that $\gamma$ is the borrow name for

[1] `{math_type T}` denotes the type of mathematical values for `T`. We come back to the *bor* wrapper shortly.

```
1  fn get_mut_client() {
2    let mut x = vec![100, 200, 300];
3    let xr: &/* 'a */mut i32 = x.get_mut(1).unwrap();
4    assert!(*xr == 200);
5    *xr = 42;
6    assert!(*x.get_mut(1).unwrap() == 42);
7  }
8  #[rr::params("xs" : "list (bor {math_type T})", "γ" : "gname", "i" : "Z")]
9  #[rr::args("(#xs, γ)", "i")]                          #[rr::exists("γi")]
10 #[rr::returns("if i < length xs then Some (xs !!! i, γi) else None")]
11 #[rr::ensures(#iris "if i < length xs then Res γ (<[i:=*γi]>xs) else Res γ xs")]
12 fn Vec::get_mut<'a>(&'a mut self, idx: usize) -> Option<&'a mut T> {
13   if idx < self.len() { unsafe { Some (self.get_unchecked_mut(idx)) } }
14   else { None }
15 }
16 #[rr::params("xs" : "list (bor {math_type T})", "γ" : "gname", "i" : "Z")]
17 #[rr::args("(#xs, γ)", "i")]
18 #[rr::requires("i < length xs")]
19 #[rr::exists("γi")]
20 #[rr::returns("(xs !!! i, γi)")]
21 #[rr::ensures(#iris "Res γ (<[i:=*γi]> xs)")]
22 unsafe fn Vec::get_unchecked_mut<'a>(&'a mut self, idx: usize) -> &'a mut T;
```

Lifetime `'a` alive, `xr` usable, `x` unusable

Fig. 3. Implementation, specification, and client of the function `Vec::get_mut`. The implementation of the unsafe function `Vec::get_unchecked_mut` can be found in Figure 5.

the **`self`** argument, *i.e.*, the vector.) However, if the index `i` is within bounds, we do not know yet what the value of the vector will be once the lifetime `'a` ends, as its `i`-th element can be modified through the returned references—and we do not know yet how this reference will be used. We express this in the specification by updating the `i`-th element of `xs` to $*\gamma i$ (via the list update syntax `<[i:=*γi]>xs`), representing a placeholder for the final value of the reference with borrow name $\gamma i$. This placeholder will be resolved once the returned reference goes out of scope. Thus, our specification essentially describes the value of the vector relative to any modifications through the returned reference. To make this work, the element type of `xs` is enriched via *bor*, where:

$$bor\ \tau \ni g ::= \#x \mid *\gamma \qquad (x \in \tau)$$

Intuitively, $bor\ \tau$ represents a *potentially borrowed* value of mathematical type $\tau$, where $\#x$ means that the value $x$ is known, and $*\gamma$ that the value is borrowed by a reference with borrow name $\gamma$.

Going back to the example in Figure 3, calling `get_mut` on line 3 first implicitly creates a new mutable reference to `x`, updating the mathematical value of `x` to $*\gamma$. The reference is then passed to the function. After the call, we obtain `Res γ [#100; *γi; #300]` from the postcondition. Once the lifetime `'a` of `xr` ends (line line 5), we further obtain `Res γi 42`. RefinedRust combines these facts to update the mathematical value of `x` to `[#100; #42; #300]`, allowing it to prove the assertion (line 6).

## 2.4 Unsafe Functions

So far, we only considered verifying safe code. This changes when we zoom in to the verification of the *implementation* of `get_mut`, which uses the low-level function `get_unchecked_mut` (Figure 3). As the name suggests, `get_unchecked_mut` does not check for out-of-bounds accesses. It could thus exhibit undefined behavior and is marked as **`unsafe`**. To call `get_unchecked_mut` one needs to meet the precondition `i < length xs`. At the call site in `get_mut`, the **`unsafe`** block indicates that the Rust

compiler cannot check this precondition; this becomes the programmer's responsibility. Such unsafe functions with extra preconditions are fully supported by RefinedRust: the specification uses a `requires` clause to express when the function is safe to call. At the call site, checking this precondition becomes the responsibility of RefinedRust, which invokes its theory solver.

## 3 UNSAFE APIS

In the previous section we showed that RefinedRust can reason about **`unsafe`** functions through additional preconditions. The more challenging part is to verify APIs that internally use **`unsafe`** C-style raw pointers. An example of such an API that we consider is `Vec`, but the use of raw pointers is widespread in the lower levels of the Rust ecosystem—it is used for data structures (*e.g.*, `HashMap`, `LinkedList`), smart pointers (*e.g.*, `Cell`, `RefCell`, `Rc`), concurrency primitives (*e.g.*, `Mutex`), and more. C-style raw pointers provide additional flexibility, but do not obey to Rust's ownership discipline, so Rust cannot determine their use to be safe (*i.e.*, to not have undefined behavior). In RefinedRust we can verify safety and functional correctness of APIs that use raw pointers. We do this by equipping the API with a *representation invariant* that specifies the internal (pointer) structure in mathematics, and proving that each function that is part of the API interface preserves the invariant.

Existing Rust verification tools, such as Creusot and Prusti, have to assert specifications for such functions implemented with *unsafe* as axioms (and justify them with external manual proofs, such as RustHornBelt), and thus are necessarily leaving gaps in the chain of trust. A key distinction of RefinedRust is that it allows us to specify and check such functions in the same framework.

In this section we discuss the vector representation invariant (§3.1) and then verify the functions of the vector API (§3.2). Our code is based on the implementation of `Vec` in the Rustonomicon [5],which is simplified compared to the version in the Rust standard library. The memory representation is the same between both versions, so our verification captures the core challenges.

### 3.1 The Vector Representation Invariant

Figure 4 shows the definition of the `Vec` type, with annotations that we will explain below. We focus on the case that the type `T` is *not* zero-sized (we have slightly simplified the invariants accordingly), but our actual verification of `Vec` handles the zero-sized case. Internally, `Vec` is implemented using a private data structure `RawVec` that manages the vector's buffer and takes care of memory allocation. The core operation of `RawVec` is `grow`, which increases the capacity of the buffer. `Vec` is implemented as a layer on top of `RawVec`; its main job is to track which part of the buffer is initialized.

**Representation invariant of `RawVec`.** To define a type's invariant we first specify its mathematical type using the `refined_by` attribute. Our `RawVec` exposes a very low-level interface: the mathematical type of `RawVec` (line 1) consists of the memory location `b` and the currently allocated capacity `c`. The `field` attribute specifies the mathematical value of each field. The mathematical value of `cap` is the mathematical integer `c:Z`. The mathematical value of the raw pointer `ptr` is `b:loc` that denotes the memory location storing the vector's buffer. Following Rust, RefinedRust's raw pointer type does not assert ownership of the memory location (*i.e.*, the pointer may be aliased).

The main work is done by the `invariant` clauses, specifying separation logic propositions that need to hold when owning a value of the type. For `RawVec`, we first specify conditions on the capacity (line 2): the maximum offset must not exceed the maximum value representable by an `isize` (a restriction of Rust's underlying LLVM backend). The second `invariant` attribute (line 3) specifies additional ownership owned by `RawVec`, namely the permission to free the buffer. (The `#own` describes how the ownership of this assertion behaves when creating a shared reference to a `RawVec`—in this case, it becomes inaccessible.) Maybe surprisingly, the invariant of `RawVec` does not contain ownership of the buffer itself. Instead, ownership of the buffer is managed by the user of

```
1  #[rr::refined_by("(b, c)" : "(loc * Z)")]
2  #[rr::invariant("{size_of T} * c ≤ max_int isize")]
3  #[rr::invariant(#own "freeable b (c * {size_of T})")]
4  struct RawVec<T> {
5    #[rr::field("b")] ptr: *mut T,
6    #[rr::field("c")] cap: usize }
7
8  #[rr::refined_by("xs" : "list (bor {math_type T})")]
9  #[rr::exists("c" : "Z", "b" : "loc", "els" : "list (bor (option (bor {math_type T})))")]
10 #[rr::invariant(#type "b" : "els" @ "array_t (maybe_init {T}) c")]
11 #[rr::invariant("∀ i, 0 ≤ i < length xs → els !!! i = #(Some (xs !!! i))")]
12 #[rr::invariant("∀ i, length xs ≤ i < c → els !!! i = #None")]
13 #[rr::invariant("length xs ≤ c")]
14 pub struct Vec<T> {
15   #[rr::field("(b, c)")] buf: RawVec<T>,
16   #[rr::field("length xs")] len: usize }
```

Fig. 4. The `Vec` data structures and representation invariants (simplified).

`RawVec` (*e.g.*, `Vec`) and linked to the `ptr` field of `RawVec` via `b`. This simplifies the verification of `Vec` as the `Vec` operations typically directly access the buffer for reads and writes.

**Representation invariant of `Vec`.** As shown in §2.3, the mathematical type for `Vec<T>` is a list of mathematical values for type `T`, decorated with `bor` to account for the elements that are borrowed. The `exists` clause (line 9) specifies existentially quantified variables, internal to the invariant: the capacity `c` of the buffer, the concrete location `b` of the buffer storing the elements, and the elements `els` of the buffer. The vector owns the `RawVec` managing its buffer, as well as a `len` field specifying the current length of the vector. The main action happens on lines 10-12. We first assert ownership of the buffer (line 10); `#type` lets us express this by assigning a type to location `b`: `b` points to an array (represented by RefinedRust's `array_t` type) with the mathematical value `els`. The array has length `c` and stores values of RefinedRust type `maybe_init T`, expressing that elements may be potentially uninitialized. Specifically, the mathematical value `els` of the array is a list of option values. (Note that the type of `els` contains `bor` twice because for each element, either `T` or `maybe_init T` can be borrowed.) The first `length xs` elements contain the values from `xs` (line 11), while the remaining elements until the end of the capacity `c` are uninitialized (line 12), represented by `None`.

### 3.2 Verification of Vector Operations

We turn to the verification of `get_unchecked_mut` (Figure 5). Recall from §2.4 that `get_unchecked_mut` takes a mutable reference `self` to a vector as well as an index `idx`, and returns a mutable reference to the element at `idx`. While the implementation comprises just a few lines of code, the reasoning of why this operation is safe (assuming the index is within bounds) is intricate. The function *reborrows* a part of the vector by returning a mutable reference to an element, essentially providing a view into the vector. Crucially, the implementation needs to ensure that, no matter how the returned reference is used, the vector's invariant is upheld. In this, `Vec` is exemplary for the reasoning required for a whole class of *reborrowing functions* often provided by Rust APIs with non-trivial invariants.

To understand what makes this challenging, let us consider the terms and conditions that surround mutable references. The contents of a mutable reference `&'a mut T` are *borrowed* from a lender. For the duration of the loan, the borrower has exclusive access to `T`. However, this loan is limited in time: references have a lifetime `'a`, and once that lifetime ends, the lender expects to get back the full contents `T` of the borrow. Concretely, when verifying `get_unchecked_mut`, our return type

```
1  #[rr::params("xs" : "list (bor {math_type T})", "γ" : "gname", "i" : "Z")]
2  #[rr::args("(#xs, γ)", "i")]
3  #[rr::requires("i < length xs")]
4  #[rr::exists("γi")]
5  #[rr::returns("(xs !!! i, γi)")]
6  #[rr::ensures(#iris "Res γ (<[i:=*γi]>xs)")]
7  unsafe fn Vec::get_unchecked_mut<'a>(&'a mut self, idx: usize) -> &'a mut T {
8    // (initial state) {self ◁ #(#xs, γ) @ &'a_mut Vec<T> ∗ idx ◁ #i @ int_usize}
9    // (unfolded) {b ◁ #els @ array_c(maybe_init T) ∗ (∀0 ≤ i < |xs|. els !!! i = ...) ∗
10   //    self ◁ #([#(b, c); #(length xs)], γ) @ &'a_mut yoinked(Vec<T>; struct[RawVec<T>; int_usize]) ∗ ...}
11   unsafe {
12     let p = self.buf.ptr().add(idx);
13     // (get shifted ptr) {p ◁ b +_l i ∗ size_of T @ raw_ptr ∗ ...}
14     let ret = &'b mut *p;
15     // (obtain borrow) {b ◁ #(els[i := #(Some(*γi))]) @ blocked'b(array_c(maybe_init T)) ∗
16     //         ret ◁ #(xs !!! i, γi) @ &'b_mut T ∗ 'b ⊑ 'a ∗ ...}
17     ret
18     // {'a ≡ 'b ∗ self ◁ #(#(xs[i := *γi]), γ) @ &'a_mut (blocked'a(Vec<T>)) ∗ ...}
19     // (lifetime has been extended, resolve γ) {Res γ (xs[i := *γi]) ∗ ...}
20   }
21 }
```

Fig. 5. Intermediate type system states of RefinedRust when checking `get_unchecked_mut`. *The comments are included for presentation purposes, but are not required by RefinedRust.* Type assignments are denoted by $l \triangleleft x \mathbin{@} T$, stating that the memory location $l$ contains a value of *type* $T$ with mathematical value $x$.

`&'a mut T` mandates that we provide (borrowed) ownership of a single vector element. Furthermore, we have to respect the terms attached to the `self` argument (of type `&'a mut Vec<T>`) and give back all ownership of the entire vector and satisfy its invariant when `'a` ends. To achieve that, we are allowed to *rely* on the receiver of our return value (of type `&'a mut T`) to in turn respect its part of the bargain, and thus return ownership of that vector element back to us when `'a` ends.

Thus, the *core challenge* in verifying `get_unchecked_mut` is the interaction of this back-and-forth borrowing, combined with the low-level reasoning about pointer arithmetic.

**Unfolding type invariants.** Figure 5 shows the implementation augmented with annotations showing RefinedRust's type system state. In the initial state, the variables `self` and `i` have type assignments according to the function's signature (line 8), conjoined with separation logic's separating conjunction $*$. In the first step, the RefinedRust type system unfolds the `Vec` type of `self` to gain the required information for accessing its internal fields later on. In particular, this allows the type system to gain knowledge about the existentially quantified $c$, $b$, and *els*. Unfolding the `Vec` invariant is not as easy as one might expect. Since the `Vec` is only owned below a mutable reference, we need to first extract the ownership from underneath that reference, effectively altering our perspective on `self`. However, remember that when lifetime `'a` ends, all that ownership will be taken away and given back to the lender. Thus, to be allowed to extract ownership from underneath a mutable reference, we have to ensure that its lifetime does not end until ownership is re-established.

**The yoinked type.** In RefinedRust, extracting the ownership `Vec` from underneath the mutable reference is captured by the **yoinked** type. Specifically, $\&'a_{mut}(\mathbf{yoinked}(V; T))$ denotes that (some of) the reference's contents have been extracted. Here, $V$ records the original type stored in the reference, which has to be put back before its lifetime ends. $T$ specifies which ownership remains in the mutable reference; this allows "partial extraction" of the reference's contents. In the case

$$\text{MByte} \ni \beta ::= z \mid (l, n) \mid \lightning \text{ with } 0 \le z < 256 \quad \text{Val} \ni v ::= \overline{\beta} \quad \mathsf{x} \in \text{VarName}$$
$$\text{Layout} \ni \iota ::= (\mathit{size} : \text{nat}, \mathit{align} : \text{nat}) \quad \text{Lft} \ni \kappa \quad \text{Mut} \ni m ::= \text{Mut} \mid \text{Shr}$$
$$\text{PExpr} \ni p ::= \mathsf{x} \mid l \mid \texttt{load}_\iota(p) \mid p.\mathsf{i} \mid p +^{\mathsf{p}} e \quad \text{VExpr} \ni e ::= \mathsf{x} \mid v \mid e_1 \cdot_o e_2 \mid \texttt{use}_\iota(p) \mid \&^\kappa_m p \mid \texttt{call}\ e(\overline{e})$$
$$\text{Stmt} \ni s ::= \texttt{goto}\ \mathsf{b} \mid \texttt{return}\ e \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{store}_\iota(p, e);\ s \mid \texttt{startlft}(\kappa);\ s \mid \texttt{endlft}(\kappa);\ s$$

Fig. 6. Syntax of Radium (excerpted and simplified).

of the vector type, RefinedRust will yoink the representation invariant of the vector from the reference, leaving just the plain struct type (without the invariant) as $T$. The yoinked ownership of the invariant then gets added to the proof context—in particular, the type assignment for `b` (line 9).

**Borrowing a component.** After this unfolding, the actual verification starts. In the first step, the `ptr` field is offset to the i-th component, using the `add` function (line 12). Doing so requires proving that the access is within bounds of the vector's allocated memory, and as a result will also not overflow. For this, the type system interacts with the pure invariants we have specified, as well as the function's precondition that the access is within bounds of the initialized part of the vector. After `add` returns, the local variable `p` has a corresponding raw_ptr type (line 13).

The key operation of `get_unchecked_mut` is the borrow of the element referenced by the produced pointer `p` (line 14). The aliased pointer is accessed, and the type system finds the actual ownership for the object in the context. Since the pointer is within bounds, the element of type `T` can be accessed and *borrowed*, producing a new reference stored in `ret`. The new reference's lifetime is a new symbolic lifetime `'b`, which must live at most as long as `'a`, the lifetime of the full vector.

However, creating this reference does come at a cost: we have temporarily borrowed ownership of a part of the array, and it will only become accessible again once the lifetime `'b` has ended. This is expressed by the **blocked**$^{\texttt{'b}}$ $U$ type. The key feature of this type is that it can be turned into $U$ after `'b` has ended, while not allowing any operations before that.

Finally, the newly-created reference is returned. Here, the type system extends the lifetime `'b` to `'a`, making them essentially equal. This is necessary to match the desired return type `&'a mut T`.

RefinedRust also needs to show that when `'a` ends, all the ownership extracted from **`self`** can be put back where it belongs. Concretely, this requires turning **yoinked**(`Vec<T>`; $U$) back into `Vec<T>` by combining $U$ with additional ownership from the context to re-establish the invariant of `Vec`. The difficulty is that we are returning part of that ownership, so it is not possible to perform this step right now. Luckily, we only need to perform this step after `'a` has ended, at which point the borrow we returned has expired. This means we can re-assemble the invariant of `Vec` using parts that are still behind **blocked**$^{\texttt{'a}}$ (line 18).

In the final step in the proof, the mutable reference **`self`** goes out of scope. This generates a resolution: when a mutable reference with mathematical value $(x, \gamma)$ goes out of scope, Res $\gamma$ $x$ is created. In our case this generates Res $\gamma$ $(xs[i := *\gamma_i])$, which is exactly what is needed to satisfy the `ensures` clause and finish the proof.

## 4 RADIUM

Before diving into the details of the RefinedRust type system (§5), we describe *Radium*: our formalized operational semantics of a subset of Rust's MIR (Mid-level Intermediate Representation), which is based on RefinedC's Caesium semantics for C. By leveraging MIR, we follow a similar approach as existing Rust verification tools (*e.g.*, Prusti) and do not base our verification on surface-level Rust. MIR is attractive for verification as it is a simple CFG-based representation where many complicated features of surface-level Rust such as loops and match statements have been desugared.

Figure 6 provides an overview of the syntax of Radium. Radium is a CFG-based language (like MIR) where basic blocks contain statements $s$. Expressions are split into *place-expressions* $p$ for computations that result in a location $l$, and *value-expressions* $e$ for computations that result in a value $v$. Similar to CompCert [29] and Caesium, values $v$ in Radium are represented as a list of *memory bytes* $\beta$, where each memory byte can either be a normal byte (*i.e.*, an integer between 0 and 255), a location fragment (consisting of the location $l$ and the index $n$ of the fragment), or poison ☣ (*e.g.*, for uninitialized memory). For example, a 32-bit integer is represented as four normal bytes, while the location a pointer points to is represented by eight location fragments with indices 0 to 7. (RefinedRust assumes 64-bit pointers.) The Radium heap is a map from locations to memory bytes. This value and heap representation allows Radium to model the semantics of Rust in detail. In particular, integers are bounded and one can reason about the byte representation of values stored in memory. The operational semantics of Radium is based on Caesium's semantics for C, but adapted to Rust. Specifically, loads and stores are parameterized by a *layout* $\iota$ that specifies the size and alignment of the access; pointer offset operations check that the offset stays in bounds of the allocations; and binary operations on integers check for overflows. (The lifetime annotations `startlft`$(\kappa)$ and `endlft`$(\kappa)$ are automatically inserted by the frontend and described in §5.3.)

### 4.1 Layout-Parametric Verification

One key challenge of giving a detailed semantics of Rust is that the layout of types in memory might not be known. This happens for two reasons: First, Rust does not specify how the fields of structures are laid out in memory. Second, Rust features polymorphic functions, so there can be structures with an entirely unknown field type. (None of these cases affect C/Caesium, as Caesium lays out structures according to the System-V ABI and C does not support polymorphic functions).

To tackle this challenge, verification in RefinedRust is parametric over the data layouts used in the program. Since adding this parameterization directly to the operational semantics would significantly complicate an already complex semantics, we instead encode it using meta-level (*i.e.*, Coq) quantification. This allows us to keep the definition of the operational semantics itself monomorphic. When verifying a concrete closed program, we can instantiate the verification result with concrete layouts to obtain safety of the closed program (Theorem 5.1 in §5.4).

Specifically, we introduce a notion of *syntactic types* SynType that abstractly describes the data layout (see Figure 7). For instance, IntSt describes the layout of integers by their signedness and their bitwidth. More interestingly, StructSt describes a struct using a struct description *sd*, containing its name and a list of fields which each have a name and a SynType. The frontend automatically generates such a syntactic type for every Rust type.

The concrete operational semantics does not work on abstract syntactic types but rather on concrete layouts Layout, consisting of a size and alignment. For struct accesses, a StructLayout describes the location of each field in the struct. Compared to StructDesc, the order of fields in StructLayout is relevant, and explicit unnamed fields for padding are included.

Conceptually, each abstract syntactic type has a set of concrete realizations as a layout (*e.g.*, for structs, they differ in the amount of padding and the order of the fields). To model this, we define a parameterized *layout algorithm* LayoutAlg that computes a concrete layout for a given syntactic type. The layout algorithm is a partial function, *e.g.*, it will fail if the type is too big to fit into `isize::MAX`. For structs, enums, and unions, it is parameterized by a subroutine that takes the type's name and fields and returns an arbitrary (but valid) layout.

All our verification results are proven for an arbitrary instance of this layout algorithm. We just have to assume that it computes *some* valid layout (made explicit in our soundness theorem,

StructDesc ∋ $sd$ ::= ($name$ : str, $fields$ : list(str × SynType)) IntType ∋ $it$ ::= ($sign$ : bool, $bits$ : nat)
SynType ∋ $st$ ::= IntSt($it$) | StructSt($sd$) | PtrSt | ...
Layout ∋ $\iota$ ::= ($size$ : nat, $align$ : nat) StructLayout ∋ $sl$ ::= $fields\_padded$ : list(option(str) × Layout)
LayoutAlg($st$ : SynType) : option(Layout) ≜ ...

Fig. 7. Core definitions for RefinedRust's layout parameterization.

see Theorem 5.1). This also allows us to handle verification of generic functions with type parameters: we parameterize the code we verify over the generic type's syntactic type, and assume that LayoutAlg is defined for all composite types in which the type parameters appear.

### 4.2 Comparison of Radium and $\lambda_{\text{Rust}}$

We describe how Radium compares to the main previous model of Rust used in foundational verification: the $\lambda_{\text{Rust}}$ model used by RustBelt and RustHornBelt. For primitive types, $\lambda_{\text{Rust}}$ uses a high-level representation with unbounded integers that fit into a single memory location. In contrast, Radium models primitive types in more detail, with bounded integers that are spread across multiple bytes. For instance, an `i32` integer only spans one memory location in $\lambda_{\text{Rust}}$, but four in Radium. Structs in $\lambda_{\text{Rust}}$ have fixed layout, and the memory model disregards alignment so there is no need for structs to have padding between their fields. Radium uses a more detailed memory model that reflects alignment constraints and is able to represent padding, and structs are properly modeled with arbitrary but fixed layout (see §4.1). As a consequence, Radium can more accurately model the conditions that unsafe code has to satisfy, *e.g.*, that all accesses are well-aligned or that pointer offset operations via `ptr::offset` are using correctly-computed field offsets. Additionally, memory accesses in Radium are typed and check more of Rust's validity constraints than $\lambda_{\text{Rust}}$ (*e.g.*, reading padding bytes as an integer is undefined behavior in Radium). $\lambda_{\text{Rust}}$ does not support integer-pointer conversion, while Radium allows round-trip casts in some cases and correctly models Rust's `NonNull::dangling` semantics, which is used to deal with zero-sized types (*e.g.*, for handling zero-sized elements of `Vec`).

## 5 TYPE SYSTEM

This section describes how the RefinedRust type system extends the Rust type system to enable the verification of safe and unsafe Rust code. We describe RefinedRust's *value types* that correspond to Rust's types extended with mathematical refinements (§5.1) and *place types* for representing partially borrowed values (including **blocked** and **yoinked** from §3) (§5.2). We then show the type system in action on a simple example (§5.3), and conclude with its soundness theorem (§5.4).

### 5.1 Value Types

RefinedRust's *value types* match Rust's notion of types: they assign types to program values. Value types are used to describe the types of argument and return values at function call boundaries. Figure 8 shows an excerpt of RefinedRust's value types. Each value type has an associated "mathematical" type that gives the mathematical representation of its values, powering RefinedRust's functional correctness reasoning (as seen in §2.1).

Value types include the basic boolean and integer types. Integer types $\mathsf{int}_{it}$ are parameterized over their bit-width and signedness via $it$ and are used to represent, for instance, `i32`. They are refined by mathematical integers $\mathbb{Z}$, while booleans are refined by mathematical booleans $\mathbb{B}$.

As already seen in §2.2, mutable references in RefinedRust are refined by a pair of the current value $x$ and the borrow name $\gamma$ used for its resolution. Shared references are just refined by the

| Rust type | RefinedRust type | math. type | intuitive semantics |
|---|---|---|---|
| `i*/u*` | $\mathsf{int}_{it}$ | $n : \mathbb{Z}$ | integers of type $it$ with math. value $n$ |
| `bool` | bool | $b : \mathbb{B}$ | booleans with mathematical value $b$ |
| `&'a mut T` | $\&^{\text{'a}}_{\text{mut}}\ T$ | $(x, \gamma) : (bor\ \tau) \times gname$ | mut. ref. with value $x$ and borrow name $\gamma$ |
| `&'a T` | $\&^{\text{'a}}_{\text{shr}}\ T$ | $x : bor\ \tau$ | shared ref. with current value $x$ |
| `Box<T>` | box $T$ | $x : bor\ \tau$ | owned pointer with current value $x$ |
| `*mut T` | raw_ptr | $l : loc$ | a raw pointer to $l$ without ownership |
| structs/tuples | $\mathsf{struct}_{sd}\ \vec{T}$ | $\vec{x} : hlist(bor\ \vec{\tau})$ | structs/tuples with field values $\vec{x}$ |
| `[T; n]` | $\mathsf{array}_n\ T$ | $\vec{x} : list(bor\ \tau)$ | arrays of $T$ with length $n$ |
| / | $\mathsf{uninit}_{st}$ | $() : ()$ | uninitialized memory with syntype $st$ |
| / | maybe_init $T$ | $x^? : option(bor\ \tau)$ | maybe initialized instance of $T$ |
| / | $\mathsf{abstract}_{\mathbf{E}}\ T$ | $x : X_{\mathbf{E}}$ | abstraction over $T$ given by $\mathbf{E}$ |

Fig. 8. Excerpt of RefinedRust's value types, where $\tau$ is the mathematical type of $T$.

value $x$ of their contents, while raw pointers are refined by the memory location that the pointer points to (they do not contain ownership). As described in §2.3, mathematical values for nested types (like the values of references and structs) are wrapped by *bor*.

Our $\mathsf{struct}_{sd}\ \vec{T}$ type is parameterized by (1) the struct description $sd$ explained in §4.1 (we omit $sd$ when it is clear from the context), and (2) a list of types $\vec{T}$ of the fields. The mathematical type is a heterogeneous list $hlist(bor\ \vec{\tau})$ of the mathematical types of its fields (with each element wrapped by *bor*). Our $\mathsf{array}_n\ T$ type models homogeneous sequences of values of type $T$ with length $n$. It is refined by the list of mathematical values for the array's elements (again wrapped in *bor*).

Additionally, RefinedRust features the $\mathsf{uninit}_{st}$ type for representing uninitialized (*i.e.*, arbitrary) memory described by the syntactic type $st$. This type has no direct correspondence in Rust, but is used to reason about uninitialized local variables and unsafe code (*e.g.*, it describes the ownership returned by Rust's unsafe allocator APIs).

Let us consider how we can define the type of `RawVec` as presented in §3. Recall that `RawVec` has two fields: the raw pointer `ptr` to the buffer (of type `*mut T`) and the capacity `cap` (of type `usize`). Thus, RefinedRust defines `RawVec`'s basic type as RawVecStruct $\triangleq \mathsf{struct}_{sd_{\text{RawVec}}}[\text{raw\_ptr}; \mathsf{int}_{\text{usize}}]$, where $sd_{\text{RawVec}} \triangleq$ [("ptr", PtrSt); ("cap", IntSt(usize))] specifies the struct's fields and their syntactic types. RawVecStruct corresponds to the Rust `RawVec` struct, however it does not include the representation invariant given by the annotations in Figure 4. To add this invariant to RawVecStruct and thus obtain RefinedRust's `RawVec` type, we leverage the $\mathsf{abstract}_{\mathbf{E}}\ T$ type that abstracts the type $T$ (*e.g.*, by adding invariants) as described by $\mathbf{E}$. Concretely, $\mathbf{E}$ contains (1) $X_{\mathbf{E}}$, the new mathematical type of $\mathsf{abstract}_{\mathbf{E}}\ T$ (given by `refined_by`, *e.g.*, $loc \times \mathbb{Z}$ for `RawVec`), and (2) an invariant specifying additional ownership and linking everything together (given by `invariant` and `field`). If an `exists` annotation is present, these variables are existentially quantified in the invariant (*e.g.*, $b$ and $c$ for `Vec`). For example, the annotations on `RawVec` in Figure 4 define the following invariant:

$$\mathsf{inv}_{\text{RawVec}}\ (b, c)\ x \triangleq x = [\#b; \#c] * \mathsf{size}(T) \cdot c \leq max\_int(\text{isize}) * \mathsf{freeable}\ b\ (c \cdot \mathsf{size}(T))$$

## 5.2 Place Types

The attentive reader may wonder where **blocked** and **yoinked** (from §3) come in. Recall that **blocked** is used to mark memory locations that have been borrowed—so the **blocked** type only

| RefinedRust type | math. type | intuitive semantics |
|---|---|---|
| $x$ @ **place** $T$ | $bor\ \tau$ | place containing $T$ |
| $(x, \gamma)$ @ $\&^{\texttt{'a}}_{\mathbf{mut}}\ \rho$ | $bor\ (\tau \times gname)$ | mutable ref. with current value $x$ and borrow name $\gamma$ |
| $x$ @ $\&^{\texttt{'a}}_{\mathbf{shr}}\ \rho$ | $bor\ \tau$ | shared ref. with current value $x$ |
| $x$ @ **box** $\rho$ | $bor\ \tau$ | owned pointer with current value $x$ |
| $\vec{x}$ @ $\mathbf{struct}_{sd}\ \vec{\rho}$ | $hlist(bor\ \vec{\tau})$ | structs/tuples with field values $\vec{x}$ |
| $\vec{x}$ @ $\mathbf{array}_n\ T$ | $list(bor\ \tau)$ | arrays of length $n$ of type **place** $T$ |
| $x$ @ $\mathbf{blocked}^{\texttt{'a}}\ T$ | $bor\ \tau$ | blocked place containing $T$ after 'a ends |
| $x$ @ $\mathbf{yoinked}(\rho_{\text{full}}; \rho_{\text{cur}})$ | $\tau_{\text{cur}}$ | yoinked type |

Fig. 9. Excerpt of RefinedRust's place types, where $\tau$ is the mathematical type of $\rho$ or $T$, respectively.

makes sense when assigned to a location, not to a value. For this reason, **blocked** is not a value type but a *place type* that is assigned to a place (*i.e.*, a memory location, or "lvalue"), not a value.

Intuitively, place types describe what happens when we read from, write to, or borrow (*i.e.*, create a reference to) a place in memory. Place types are not part of Rust's syntax of types, but rather are RefinedRust's way to track intermediate states of Rust's type system. As a consequence, place types only appear during RefinedRust's type checking process, but not in top-level specifications.

**Place types in RefinedRust.** Figure 9 shows an excerpt of RefinedRust's place types (meta-variable $\rho$), along with their mathematical type. Like value types, the mathematical types of the place types use *bor* to denote where borrows can happen.

The place type **place** $T$ transforms the value type $T$ into a place type, stating that memory contains a value of type $T$. The place type $\mathbf{blocked}^{\texttt{'a}}\ T$ states that the place is blocked for lifetime 'a (since it has been borrowed) and will have type **place** $T$ after 'a ends. We also have already seen the place type $\mathbf{yoinked}(\rho_{\text{full}}; \rho_{\text{cur}})$ which states that the place originally had the place type $\rho_{\text{full}}$, but currently has the place type $\rho_{\text{cur}}$ because ownership has been yoinked.

The primitive reference and struct types appear not only as value types, but also have a corresponding place type. This is because their corresponding Rust types support place accesses below them. There is a clear correspondence between these value types and their place types:

$$\mathbf{place}(\&^{\kappa}_{\text{mut}} T) \equiv \&^{\kappa}_{\mathbf{mut}}(\mathbf{place}\ T) \qquad \mathbf{place}(\text{struct}_{sd}\,\vec{T}) \equiv \mathbf{struct}_{sd}(\overrightarrow{\mathbf{place}\ T})$$

The first equivalence states that a place containing a value of the mutable reference type $\&^{\texttt{'a}}_{\text{mut}} T$ is equivalent to a mutable reference place $\&^{\texttt{'a}}_{\mathbf{mut}}(\mathbf{place}\ T)$ with the referenced place containing a value at type $T$. Similar "unfolding equations" hold for shared references, arrays, and other types. The expressiveness of these "simple" place types becomes clear when combining them with $\mathbf{blocked}^{\kappa}\ T$: They allow creating types like $\mathbf{struct}[\mathbf{blocked}^{\texttt{'a}}(\text{int}_{\text{i32}}); \mathbf{place}(\text{int}_{\text{i32}})]$, representing a structure where the first field has been borrowed. In §5.3 we explain this interplay using an example.

Place types $\rho$ are assigned to memory locations $l$ via the place type assignment $l \lhd x$ @ $\rho$, where $x$ is the mathematical value stored in the place. We already saw place types in action in §3.2.

### 5.3 RefinedRust's Type System in Action

Let us now explain some of RefinedRust's typing rules by following the RefinedRust type checker through the verification of the Radium code in Figure 10.[2] First, we go over the code, ignoring the comments that show the state of the type system. The code snippet creates a tuple z of two

[2]For space reasons, there are some technical details that we are omitting here: place type assignments have an additional parameter that is needed in some corner cases, and we need some extra machinery to deal with "later" modalities [44].

```
1  // {z ◁ place(uninit_(i32,i32)) ∗ zr ◁ place(uninit_PtrSt)}
2  let mut z = (0, 1);
3  // {z ◁ #[#0; #1] @ place(struct[int_i32; int_i32]) ∗ zr ◁ place(uninit_PtrSt)}
4  startlft 'a;
5  // {'a alive ∗ z ◁ #[#0; #1] @ place(struct[int_i32; int_i32]) ∗ zr ◁ place(uninit_PtrSt)
6  let zr = &'a mut z.0;
7  // {'a alive ∗ z ◁ #[∗γ; #1] @ struct[blocked'a(int_i32); place(int_i32)] ∗ zr ◁ #(#0, γ) @ place(&'a_mut(int_i32))}
8  *zr = 42;
9  // {'a alive ∗ z ◁ #[∗γ; #1] @ struct[blocked'a(int_i32); place(int_i32)] ∗ zr ◁ #(#42, γ) @ place(&'a_mut(int_i32))}
10 endlft 'a;
11 // {'a dead ∗ z ◁ #[∗γ; #1] @ struct[blocked'a(int_i32); place(int_i32)] ∗ zr ◁ place(uninit_PtrSt) ∗ Res γ 42}
12 assert!(z.0 == 42 && z.1 == 1));
13 // {'a dead ∗ z ◁ #[#42; #1] @ struct[place(int_i32); place(int_i32)]) ∗ zr ◁ place(uninit_PtrSt)}
```

Fig. 10. A simple type checking example involving a mutable borrow from a pair.

integers, and then mutably borrows its first component as `zr`. Here, the lifetime `'a` of the reference is created explicitly using the `startlft 'a` annotation: the RefinedRust type system relies on lifetime annotations for references as hints. (These annotations are inserted by the frontend translating Rust into Radium.) The reference `zr` is then used to write a new value, 42, into the first component. After that, the lifetime `'a` is ended with an annotation by the frontend. In the last step, the code asserts that the write through the reference updated the tuple as expected.

Now we consider what happens to the types. In the beginning, both `z` and `zr` are uninitialized. Then, line 2 initializes `z` with `(0, 1)` which updates the type of `z` to $[\#0; \#1]$ @ $\mathsf{struct}[\mathsf{int}_{\mathsf{i32}}; \mathsf{int}_{\mathsf{i32}}]$ (converted to a place type using **place** $T$). Next, RefinedRust processes the `startlft` annotation on line 4, and allocates a new symbolic lifetime for `'a`. The fact that the lifetime `'a` is alive is tracked by the `'a` alive assertion in line 5.

**Checking mutable borrows.** Now, let us consider the creation of the mutable borrow of the first field of `z` (line 6) in detail. RefinedRust's general procedure CHK-MUT-BOR for type checking a mutable borrow of expression $e$ at lifetime `'a` is provided in Figure 11. In the case of our example, this procedure is called with `z.0` for $e$. CHK-MUT-BOR first decomposes this expression into a base location $l_o$, in this case $z$, and a sequence of *place accesses* $\mathcal{P}$ to it, in this case $[\mathsf{Field}(\text{“0”})]$. We then find the type assignment for $l_o$ in the context (line 3)—here the place type $\rho_o$ of $z$ is **place**$(\mathsf{struct}[\mathsf{int}_{\mathsf{i32}}; \mathsf{int}_{\mathsf{i32}}])$ with the mathematical value $x_o = \#[\#0; \#1]$.

Then, CHK-PLACE-ACCESS (explained below) is called (line 4) to check that the sequence of place accesses $\mathcal{P}$ is valid for the given type of $l_o$. CHK-PLACE-ACCESS also determines the resulting memory location $l_i$ and type assignment $l_i \lhd x_i \mathbin{@} \rho_i$ for $l_i$. Furthermore, $k_{\text{min}}$ describes the minimum ownership along the path, *i.e.*, whether the place is fully owned (Owned), or we passed below a shared ($\mathsf{Shared}_\kappa$) or mutable ($\mathsf{Uniq}_\kappa$) reference, as this determines what operations we can perform on the resulting place. Finally, $\rho[\cdot]$ will be explained below. In this case, the single place access $\mathsf{Field}(\text{“0”})$ needs to be made to $z$, and we obtain $l_i = z\ \mathsf{AtField}_{(\mathsf{i32},\mathsf{i32})}\ \text{“0”}$ with the type assignment $l_i \lhd \#0 \mathbin{@} \mathbf{place}(\mathsf{int}_{\mathsf{i32}})$ (*i.e.*, $x_i = \#0$ and $\rho_i = \mathbf{place}(\mathsf{int}_{\mathsf{i32}})$). The AtField computes the offset of field “0” in the tuple. Since the access does not go below a reference, $k_{\text{min}}$ is Owned. In the next step, the procedure STRATIFY is used to bring the type into the shape $\#x_i \mathbin{@} \mathbf{place}\ T_i$. This is already the case in the example, and we have $T_i = \mathsf{int}_{\mathsf{i32}}$ and $x_i = 0$. We discuss the STRATIFY procedure in more detail below.

Now that we have obtained the place that is borrowed, we are ready to create the reference. First of all, we have to check that the minimum ownership mode $k_{\text{min}}$ along the accessed path allows mutable borrows—here, this is the case, since $k_{\text{min}} = \text{Owned}$ (line 7). Next, we create a

1: **procedure** CHK-MUT-BOR($e$, 'a)
2: $(\mathcal{P}, l_o) \leftarrow$ DECOMPOSE-EXPRESSION($e$)
3: $(x_o, \rho_o) \leftarrow$ FIND-ASSIGNMENT-FOR($l_o$) — find type assignment $l_o \lhd x_o \,@\, \rho_o$
4: $(l_i, x_i, \rho_i, k_{\min}, \rho[\cdot]) \leftarrow$ CHK-PLACE-ACCESS($l_o, x_o, \rho_o, \mathcal{P}$)
5: — apply place accesses $\mathcal{P}$ to $l_o$, resulting in $l_i \lhd x_i \,@\, \rho_i$
6: $(x_i, T_i) \leftarrow$ STRATIFY($l_i, x_i, \rho_i, k_{\min}$) — stratify to $l_i \lhd \#x_i \,@\,$ **place** $T_i$
7: **assert**($k_{min}$ allows mutable borrow at 'a)
8: $\gamma \leftarrow$ CREATE-BORROW-NAME($x_i$)
9: $(x_o', \rho_o') \leftarrow$ FILL-PCTX($\rho[\cdot], *\gamma,$ **blocked** $^{'a}$ $T_i$) — new type for $l_o$: $l_o \lhd x_o' \,@\, \rho_o'$
10: ADD-TO-CONTEXT($l_o \lhd x_o' \,@\, \rho_o'$) — release updated ownership
11: **return**$((\#x_i, \gamma) \,@\, \&^{'a}_{\text{mut}}\, T_i)$ — return type of expression
12: **procedure** CHK-PLACE-ACCESS($l_o, x_o, \rho_o, \mathcal{P}$)
13: $(k_{\min}, l_i, x_i, \rho_i, \rho[\cdot]) \leftarrow (\text{Owned}, l_o, x_o, \rho_o, \cdot)$
14: **while** $(a :: \mathcal{P}) \leftarrow \mathcal{P}$ **do**
15: $x_i \leftarrow$ USE-BORROW-RESOLUTION($\rho_i, x_i$) — Use resolutions at the head
16: $\rho_i \leftarrow$ PLACE-UNFOLD-HEAD($\rho_i$) — Unfold **place**$T$ at the head, if necessary
17: **match** $a, \rho_i$ **with**
18: **case** Field($f$), **struct**$_{sd}$ $\vec{\rho}$:
19: **assert**(field $f$ is a field of $sd$)
20: $(\rho_f, x_f) \leftarrow (\vec{\rho}!!f, x_i!!f)$ — look up the type $x_f \,@\, \rho_f$ of the field
21: $\rho[\cdot] \leftarrow \ldots$
22: $(k_{\min}, l_i, x_i, \rho_i) \leftarrow (k_{\min}, l_i\ \text{AtField}_{sd}\ f, x_f, \rho_f)$
23: **case** Deref, $\&^{\kappa}_{\mathbf{mut}}\rho$: ...
24: **return**$(l_i, x_i, \rho_i, k_{\min}, \rho[\cdot])$

Fig. 11. Procedure for type checking mutable borrows and place accesses.

new borrow name $\gamma$ (line 8). This allows us to create the mutable reference, in this case of type $(\#0, \gamma) \,@\, \&^{'a}_{\text{mut}}(\text{int}_{i32})$, which is returned in line 11.

Before we finish up creating the mutable reference, there is however another question that needs to be answered: what is the new type of $l_o$? Intuitively, we need to block the place we borrow until lifetime 'a ends—*i.e.*, $l_i$ should have type $*\gamma \,@\,$ **blocked** $^{'a}$ $T_i$. Now we just need to translate this type for $l_i$ to a type for $l_o$. This is the purpose of $\rho[\cdot]$: $\rho[\cdot]$ is a *place type context* that describes the type of $l_o$ with a hole for the new type of $l_i$. Concretely, we have $\rho[\cdot] = \#[\cdot; \#1] \,@\,$ **struct**$[\cdot;$ **place**$(\text{int}_{i32})]$. FILL-PCTX on line 9 fills $\rho[\cdot]$ with the new type for $l_i$, obtaining the new place type $\rho_o' =$ **struct**$[$**blocked** $^{'a}(\text{int}_{i32}); $ **place**$(\text{int}_{i32})]$ with mathematical value $x_o' = \#[*\gamma; \#1]$ for $l_o$. This type assignment for $l_o$ is then added back to the context in line 10.

**Checking place accesses.** Now, let us consider CHK-PLACE-ACCESS in Figure 11 that is called on line 4 of CHK-MUT-BOR. In our example, it is called with $l_o = z$, $x_o = \#[\#0; \#1]$, $\mathcal{P} = [\text{Field}(\text{“0”})]$, and $\rho_o =$ **place**$(\text{struct}[\text{int}_{i32}; \text{int}_{i32}])$. CHK-PLACE-ACCESS applies the sequence of accesses $\mathcal{P}$ to the place type, while keeping track of the minimum permission $k_{\min}$ along the way. For each access, it first uses resolutions at the head (line 15) to ensure that the current type assignment is of the form $l_i \lhd \#x_i \,@\, \rho_i$. Then, for $\rho_i =$ **place** $T_i$, it unfolds the place type at the head using the already-discussed equivalences (line 16). In our example, it unfolds $\rho_i$ to **struct**$[$**place**$(\text{int}_{i32})$; **place**$(\text{int}_{i32})]$. Then, CHK-PLACE-ACCESS matches on the next place operation and current place type. In the case of a field access to field $f$ of a struct type, we first check that $f$ is a valid field for the struct (line 19). Then, we look up the type $\rho_f$ and mathematical value $x_f$ (line 20) for $f$, and update $\rho[\cdot]$ (omitted) and the iteration variables. Finally, after having processed all place accesses, the current iteration variables are returned.

**Ending lifetimes.** Let us continue with the example in Figure 10. The write to the mutable reference `zr` on line 8 updates the current value of the mutable reference to 42, importantly using the fact that `'a` is alive to justify the write. The `endlft` instruction on line 10 ends the lifetime `'a`, replacing the `'a` alive assertion in the context with the `'a` dead assertion. In addition, we deinitialize all mutable references that are now inaccessible and extract resolutions about their final value. In our example, we get the resolution Res $\gamma$ 42 about `zr`'s final value. Note that we do not yet unblock the inaccessible component `z.0`: this is done lazily on the next access.

**Reading and stratification.** Finally, line 12 reads from the two components of `z` to assert their final values. The reads happen in a way that is conceptually similar to the mutable borrow before (Figure 11), as both are place accesses — only instead of creating a new mutable borrow in the end, we create a copy of the integers in the two components. However, the stratification step is more interesting here: since the current place type of `z.0` is $\mathbf{blocked}^{\texttt{'a}}(\mathsf{int}_{\mathsf{i32}})$, STRATIFY will unblock the place by using the fact that `'a` is dead and then use the resolution Res $\gamma$ 42 to update the $*\gamma$ to #42.

### 5.4 Soundness

We define the RefinedRust type system by building a semantic model in the Iris separation logic framework [48, 18]. This means that all RefinedRust types and typing judgments are defined as predicates in separation logic, and each typing rule is phrased as a separation logic lemma and proven sound against these predicates. An important aspect of the RefinedRust semantic model is the use of RustBelt's *lifetime logic* [18, 15], which extends separation logic with a notion of "borrowing", where ownership of an arbitrary separation logic proposition can be split into ownership *during* a lifetime, and ownership *after the end* of that lifetime. This feature is at the core of RustBelt's model of references to split the ownership of the borrowed value between the borrower and the lender. RefinedRust extends the lifetime logic to model place types. Concretely, to enable introducing the **blocked** type below mutable references (as *e.g.*, in `get_unchecked_mut` in §3), we introduce a notion of *pinned borrows* that allows temporarily weakening the type under a mutable reference. Details can be found in the supplementary material [11]. Our formalization of the type system consists of around 21k lines of specification and 14k lines of proof. Using Iris's soundness theorem, we obtain a top-level soundness theorem for RefinedRust.

THEOREM 5.1 (ADEQUACY). *Let $\mathcal{F}$ be a "main" function for which the RefinedRust type system (instantiated with a layout algorithm that can layout all types used by the program) has verified a type corresponding to the Rust type* `() -> ()`*. Then $\mathcal{F}$ executes safely, i.e., it will neither cause undefined behavior nor cause a panic.*

RefinedRust's adequacy statement follows the standard structure of adequacy statements for Iris-based type systems like RustBelt, RustHornBelt, and RefinedC: It states that a well-typed (*i.e.*, verified) closed program has no undefined behavior and no panics. To understand the guarantees provided by this theorem, one has to consider that RefinedRust is compositional: for instance, if a function promises in its postcondition that it returns an even integer, then compositionality ensures that we can form a larger closed program that checks whether the integer is actually even, and panics otherwise. Adequacy on that larger program then says that the panic can never happen, therefore implying that the postcondition is correct.

## 6 USING REFINEDRUST FOR VERIFICATION

To demonstrate that it is feasible to use the RefinedRust type system for verifying real Rust code, we have implemented a type checker for RefinedRust in the Coq proof assistant. The RefinedRust implementation uses the Lithium separation logic engine [40, 41] to automatically apply RefinedRust typing rules, and tries to solve as many pure side conditions posed by the typing rules as possible.

**Frontend.** The RefinedRust frontend translates Rust's MIR code into Radium. The frontend is implemented as a plugin to the `rustc` compiler that runs after the MIR code has been generated and after Rust's borrow checker has run successfully. For the most part, the frontend directly maps MIR to Radium. However, in addition it generates hints for the RefinedRust type system relating to lifetimes of references, extracted from the experimental Polonius borrow checker [36] (slated to eventually replace the current stable borrow checker). A non-trivial aspect of this translation is to align Polonius' notion of *loans* with the concept of *lifetimes* used in RustBelt and inherited by the RefinedRust type system. We reuse a few utility functions from the Prusti implementation [3], in particular to extract information about lifetimes and references from Polonius. In addition to the code, the RefinedRust frontend translates the RefinedRust annotations into types and specifications and generates lemmas stating that the code of a function satisfies its specification. The proofs of these lemmas invoke the RefinedRust type checker implemented using Coq proof automation.

**Trusted Computing Base.** As RefinedRust directly applies the RefinedRust typing rules proved sound in Coq and the resulting proofs are checked by Coq, all proofs done with RefinedRust are fully foundational. RefinedRust's trusted computing base consists of Radium, which we assume to provide a reasonably accurate model of the Rust operational semantics, as well as the frontend, which needs to correctly translate definitions to Radium, and the top-level safety statement (in addition to the kernel of the Coq proof assistant and its infrastructure). The additional lifetime annotations generated by the frontend need not be trusted—our verification merely uses them as hints. One also does not need to trust the RefinedRust type system, the lifetime logic, or its implementation in Iris, because one can use Theorem 5.1 to obtain a correctness statement that just refers to the operational semantics of Radium (without referring to the type system or Iris).

**Evaluation.** Next to the small example functions given in this paper, we have evaluated RefinedRust's ability to verify unsafe code by verifying core parts of the `Vec` API as presented in the Rustonomicon [5]. Specifically, we have verified the following parts of the `Vec` API: `new`, `push`, `pop`, `get_unchecked`, `get`, `get_mut_unchecked`, `get_mut`, and `len` (in addition to internal accessor functions). We also verified `RawVec` with its `new` and `grow` functions as it is used internally by `Vec` (see §3.1), and shims for pointer manipulation and allocation (*e.g.*, `alloc::{alloc, dealloc, realloc}`).

The `Vec` code in the Rustonomicon is simplified compared to the standard library version in a few places. First of all, it is not parameterized over the allocator that is used for memory allocation (instead using Rust's global allocator). Secondly, `get` and `get_mut` on the standard library `Vec` implementation work by converting a vector to a slice and then using the `get`/`get_mut` methods on slices, while in the Rustonomicon implementation they are directly implemented on `Vec`. Additionally, we have modified the Rustonomicon version by writing wrappers for the low-level memory allocation APIs that the code uses. The Rust standard library memory allocation functions are very platform-specific and use features that RefinedRust currently does not support.

The functions we have verified for the `Vec` and `RawVec` APIs range between 3 and 20 lines of code. In total, these APIs are implemented with 120 lines of code (measured with `tokei`). Our annotations for representation invariants and function specifications add an additional 76 lines of code. The Radium code for `Vec` comprises roughly 1200 lines of (automatically generated) Coq code. A large part of this blow-up comes from the lowering of (surface) Rust to MIR by the Rust compiler, which induces a significant overhead by desugaring operations and introducing temporary variables: the MIR code that the RefinedRust frontend takes as input comprises 900 lines of code. The additional Radium lines come from annotations for lifetimes and typing hints, as well as for local variable declarations. The code size blow-up, the complexities of RefinedRust's type system for handling reference types, and the generation (and checking) of the foundational proof in Coq make verification performance intensive. In total, the verification of the whole `Vec` API takes about 6 minutes wall time (and 22 minutes CPU time) on a recent Apple M1 Max processor. In addition to specifications for individual

Table 1. Comparison of related work (func. correct. = proving functional correctness; foundational = foundational proofs in a proof assistant; unsafe: supports verifying unsafe code; mem. = has a detailed memory model that captures UB when working with (raw) pointers (*e.g.*, alignment, out-of-bound offsets, zero-sized types); automated = provides automated verification and takes real Rust programs as input).

| | func. correct. | foundational | unsafe | mem. | automated |
|---|---|---|---|---|---|
| **RustBelt** [18] | ○ | ● | ● | ○ | ○ |
| **RustHornBelt** [31] | ● | ● | ● | ○ | ○ |
| **Creusot** [9] | ● | ○ | ○ | ○ | ● |
| **GillianRust** [55] | ● | ○ | ● | ● | ● |
| **Flux** [28] | ● | ○ | ○ | ○ | ● |
| **Verus** [27] | ● | ○ | ◐ | ○ | ● |
| **Prusti** [3] | ● | ○ | ○ | ○ | ● |
| **Aeneas** [12] | ● | ◐ | ○ | ○ | ◐ |
| **RefinedRust** | ● | ● | ● | ● | ◐ |

functions, the verification uses around 80 lines of manually-proved Coq theory for reasoning about `Vec`'s representation invariant. For example, for `Vec::pop`, the type system makes roughly 3.000 automatic steps for ownership reasoning, and generates 100 pure Coq side conditions. Of these side conditions, all but five are solved automatically. Solving the remaining side conditions for `pop` requires about 20 lines of manual Coq proofs.

## 7 RELATED WORK

There is a long line of work on verifying low-level pointer manipulating code, especially in the context of C [8, 24, 13, 1, 10, 41, 37, 35], and on the theory of ownership/region-based type systems similar to/or preceding Rust [7, 51, 49, 14, 6]. We now zoom in on tools for verifying Rust programs.

Table 1 compares recent approaches for verifying Rust programs based on the aspects focussed on by this work. As Table 1 shows, RefinedRust is the first tool that supports automated and foundational functional correctness proofs for unsafe Rust code against a detailed memory model. In particular, most existing automated verification tools do not support reasoning about unsafe code and do not provide foundational proofs. On the other hand, RustBelt and RustHornBelt rely on manual verification and translation of Rust code while also using a significantly simpler operational model than RefinedRust (see §4.2). Let us now compare with the different approaches in detail.

**RustHorn, Creusot, RustHornBelt, and GillianRust.** RustHorn [32] is an approach to functionally verifying safe Rust programs by generating an encoding of them in terms of Horn clauses, building on the key insight that purely safe Rust programs are essentially functional. RustHorn uses this insight to encode mutable references as a pair of the current value and a prophecy variable for the final value (which also inspired RefinedRust's encoding).

This approach has been implemented in a practical tool, Creusot [9]. Creusot supports a wide range of Rust language features (*e.g.*, traits and closures) and has been used to verify intricate case studies, such as the verification of an optimized SAT solver [43].

However, RustHorn's approach inherently cannot be used to reason about pointer-manipulating unsafe code. The best that can be done is reason about safe code calling such unsafe code, and even that only works if the unsafe code has a purely functional specification. That specification is then assumed as an axiom by RustHorn/Creusot.

RustHornBelt [31] can be used to formally verify those axioms in Coq, and shows that the core verification technique of these tools is sound. However, RustHornBelt is closely based on RustBelt, so proofs in RustHornBelt share some of the same limitations: compared to RefinedRust, significantly more manual work is required, both in translating the original Rust code into a model suitable for formal verification, and in actually carrying out the proof. Furthermore, the connection between RustHorn/Creusot and RustHornBelt has not been made formal even on paper. In contrast, RefinedRust demonstrates a methodology for verifying safe and unsafe code in a unified semi-automated framework.

Another approach to verifying Creusot specifications of unsafe code is explored by GillianRust [55] (developed concurrently with RefinedRust). GillianRust is a non-foundational Rust verifier built on the Gillian verification framework [42, 30]. GillianRust enables reasoning about unsafe Rust code using an axiomatization of RustBelt's lifetime logic and achieves a high degree of automation thanks to its SMT-based verification (*e.g.*, demonstrated on the verification of a doubly-linked list).

**Flux.** Flux [28] extends Rust's type system with refinement types for functional verification, inspired by the "liquid types" approach [38]. To handle writes to mutable references, Flux introduces a notion of "strong references" that permit the reference's type to change (*i.e.*, allow strong updates) by tracking the exact location that is borrowed, reminiscent of RefinedC's &own type described in §1. Flux leverages these strong borrows to build a lightweight and highly automated verification tool that can automatically synthesize refinements and loop invariants. In contrast, RefinedRust requires significantly more annotations and proof guidance, and handles a smaller subset of Rust. However, Flux is limited in expressivity: it targets the verification of safe code, while RefinedRust can also verify unsafe code. In particular, Flux cannot reason about low-level pointer manipulation, so none of the methods of `Vec` we verified could be verified in Flux (already the safety invariant on the `Vec` structure in Figure 4 is inexpressible in Flux, as it requires specifying custom ownership over memory). Instead, `Vec` is axiomatized in Flux with a weaker interface that tracks only the length of the vector instead of its contents (and so, the Flux specification of `get_mut` (Figure 3 in §2.3) also does not link the returned reference to the contents of the vector).

**Verus.** Verus [27] is a Rust verifier that leverages Rust itself as the specification and proof language. As proofs are checked by an SMT solver and rely on Rust's type checker (including the borrow checker) for soundness, proofs in Verus are not foundational. Verus is more mature and supports a larger subset of safe Rust than RefinedRust, and even supports some patterns that would traditionally require unsafe code. Verus's support for unsafe code works by providing abstractions over raw pointers that are safe in conjunction with side conditions checked by the SMT solver, essentially delegating ownership reasoning to Rust's ownership type system. This is powerful, but any ownership reasoning requires using dedicated Rust types that encode this ownership. Thus, Verus cannot verify the `Vec` implementation that is written with raw pointers directly. Moreover, any of these abstractions (as well as the Rust type checker) have to be trusted. Moreover, Verus currently cannot verify *reborrowing functions* that return a mutable reference like the function `Vec::get_unchecked_mut` we verify in §3.

**Prusti.** Prusti [3] is a Rust verification tool based on the Viper [34] verification infrastructure. Prusti uses Rust type signatures to infer the requisite ownership in pre- and postconditions (which has served as inspiration of RefinedRust's handling of safe code). Thanks to Viper's SMT-based solver, Prusti provides a high degree of automation. To model Rust's mutable reference types, Prusti has a notion of *pledges*, describing assertions that hold once the lifetime of a reference ends. Pledges are similar in flavor to RefinedRust's borrow names, but less flexible. For instance, Prusti cannot state the `Vec::get_mut` specification shown in §2.2 since it does not support mutable references inside of `Option`. Prusti does have a model of mutable state (albeit a more high-level one than RefinedRust),

so it would in principle support reasoning about raw pointers. However, it has not been used yet for unsafe verification. In particular, specifications for Rust's vectors are asserted as axioms.

**Aeneas.** Aeneas [12] is a verification toolchain for safe Rust based on a translation to a borrow calculus with a pure, functional semantics. Programs in this calculus can be translated into multiple provers, *e.g.*, F* or Coq, which are then used to reason about the generated code. The assurances and automation depend on the chosen backend.

**Other tools for increasing Rust assurances.** Apart from deductive verification, other tools have been proposed for finding bugs or verifying Rust programs. Most of these tools have a lower barrier to entry for programmers than the deductive verification tools, but are more restricted in terms of expressivity or the provided assurances.

KRust [52] and RustSEM [22] formalize Rust's semantics in the K framework [39]. KRust formalizes core parts of safe Rust (including some parts not handled by RefinedRust like closures), but does not formalize unsafe Rust. RustSEM formalizes more extensive parts both of safe and unsafe Rust, but at a comparatively high-level (*e.g.*, the memory model does not reflect the byte-level representation of values). The authors of RustSEM use the K framework's ability to derive a verification tool from the semantics and use it to verify some unsafe code, including four functions of Rust's `VecDeque` API. However, they only verify that the head, tail, and capacity of the `VecDeque` are staying consistent with each other; they do not verify a full functional correctness specification like our `Vec` case study. Furthermore, their framework does not support unbounded heap fragments, so the verification is limited to `VecDeque`'s of length 16, making it more akin to bounded model checking.

Miri [33] is an interpreter for Rust's MIR intermediate representation that can check for many forms of undefined behavior in unsafe Rust code. Thanks to its ease of use, Miri has become the de-facto tool for programmers of unsafe Rust to check their code for compliance with Rust's rules, and it has been successful in uncovering bugs in Rust's standard library. Due to Miri's focus on checking individual executions, it is limited to bug-finding as opposed to verification.

Kani [23] is a bounded model checker for Rust, which can reason about all program executions (if a computable bound on the execution length can be found). Kani supports raw pointers with a low-level memory model, and has thus turned into a valuable tool for programmers of unsafe Rust to gain basic assurances. Kani has some limitations inherent to bounded model checking: its expressiveness around loops is limited, requiring easily computable loop bounds, and it cannot express modular Hoare-style specifications (with preconditions and postconditions).

## 8 FUTURE WORK

The RefinedRust type system represents a crucial first step towards high-assurance verification of Rust programs with both safe and unsafe code. Our prototype implementation of RefinedRust in Coq enables the first foundational functional correctness proofs of real Rust code with respect to a realistic operational semantics. In future work, we would like to improve the user friendliness, verification times, and handling of pure side conditions. These aspects are orthogonal to the foundations of the type system. For example, we plan to explore if a recently developed solver for arrays in Coq [53] could be integrated to discharge the side conditions in our `Vec` case study. RefinedRust might also provide a basis for standalone verification tools, similar to the way foundational logics for weak-memory verification have been axiomatized in the Viper framework [46] (that also underlies Prusti). Another avenue for future work is to expand RefinedRust's support for advanced features of Rust, such as closures, *e.g.*, by taking inspiration from Wolff et al. [54]. Finally, while Radium, our formal model of Rust, is strictly more accurate than the models used by prior deductive verification tools, there are aspects of Rust we do not model; most of them do not even have an official specification yet. The ongoing development of a normative specification for Rust [16] could provide essential guidelines to improve Radium.

## ARTIFACT AVAILABILITY

The supplementary material, including our implementation of RefinedRust and the formalization of RefinedRust's type system, is available online [11].

## ACKNOWLEDGMENTS

We thank Rodolphe Lepigre and Deepak Garg for helpful discussions in early stages of the project, as well as the reviewers for their constructive feedback. We thank Amazon Web Services for a research award supporting the work on this project. The second author was supported by a Google PhD fellowship during his work on this project.

## REFERENCES

[1] Andrew W. Appel. 2014. *Program Logics for Certified Compilers*. https://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers
[2] Matt Asay. 2020. Why AWS loves Rust, and how we'd like to help. https://aws.amazon.com/de/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/. Last accessed 07 October 2021.
[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573
[4] The Rustonomicon authors. 2023. The nullable pointer optimization. https://doc.rust-lang.org/nomicon/ffi.html#the-nullable-pointer-optimization Last accessed 16 Nov 2023.
[5] The Rustonomicon authors. 2023. Vector implementation. https://doc.rust-lang.org/nomicon/vec/vec-final.html Last accessed 16 Nov 2023.
[6] David G. Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*. 292–310. https://doi.org/10.1145/582419.582447
[7] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *OOPSLA*. https://doi.org/10.1145/286936.286947
[8] 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 *TPHOLs (LNCS, Vol. 5674)*. 23–42. https://doi.org/10.1007/978-3-642-03359-9_2
[9] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a Foundry for the Deductive Verication of Rust Programs. In *ICFEM 2022 - 23th International Conference on Formal Engineering Methods (LNCS)*. Springer Verlag, Madrid, Spain. https://hal.inria.fr/hal-03737878
[10] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. 646–661. https://doi.org/10.1145/3192366.3192381
[11] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2023. RefinedRust: Technical Documentation and Coq Development. https://doi.org/10.5281/zenodo.10912439 Project website: https://plv.mpi-sws.org/refinedrust/.
[12] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647
[13] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (LNCS, Vol. 6617)*. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4
[14] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *USENIX ATC*. 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html
[15] Ralf Jung. 2020. *Understanding and evolving the Rust programming language*. Ph. D. Dissertation. Universität des Saarlandes, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647
[16] Ralf Jung. 2023. MiniRust. https://github.com/RalfJung/minirust Last accessed 16 Nov 2023.
[17] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109
[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154
[19] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

[20] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[21] 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. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

[22] Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 2020. An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing. *CoRR* abs/1804.07608 (2020). http://arxiv.org/abs/1804.07608

[23] The Kani Developers. 2022. The Kani Rust Verifier. https://github.com/model-checking/kani Last accessed 01 October 2022.

[24] 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 *SOSP*. 207–220. https://doi.org/10.1145/1629575.1629596

[25] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP (LNCS, Vol. 10201)*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

[26] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

[27] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037

[28] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *PACMPL* 7, PLDI (2023), 1533–1557. https://doi.org/10.1145/3591283

[29] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2*. Technical Report RR-7987. Inria. https://hal.inria.fr/hal-00703441

[30] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 827–850. https://doi.org/10.1007/978-3-030-81688-9_38

[31] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. ACM. https://doi.org/10.1145/3519939.3523704

[32] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 484–514.

[33] The Miri Developers. 2015. Miri: An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri Last accessed 24 October 2022.

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

[35] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*. ACM, 440–451. https://doi.org/10.1145/2594291.2594325

[36] Polonius. 2018. Rust Polonius. https://github.com/rust-lang/polonius. Last accessed 03 October 2021.

[37] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying Systems C Code with Separation-Logic Refinement Types. *Proc. ACM Program. Lang.* 7, POPL (2023), 1–32. https://doi.org/10.1145/3571194

[38] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*. 159–169. https://doi.org/10.1145/1375581.1375602

[39] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012

[40] Michael Sammler. 2023. *Automated and Foundational Verification of Low-Level Programs*. Ph. D. Dissertation. Universität des Saarlandes, Saarbrücken, Germany. https://doi.org/10.22028/D291-41316

[41] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *PLDI*. 158–174. https://doi.org/10.1145/3453483.3454036

[42] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *PLDI*. ACM, 927–942. https://doi.org/10.1145/3385412.3386014

[43] Sarek Høverstad Skotåm. 2022. *CreuSAT, Using Rust and Creusot to create the world's fastest deductively verified SAT solver*. Master's thesis. University of Oslo. https://www.duo.uio.no/handle/10852/96757

[44] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631
[45] Jeff Vander Stoep and Stephen Hines. 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html. Last accessed 07 October 2021.
[46] Alexander J. Summers and Peter Müller. 2018. Automating Deductive Verification for Weak-Memory Programs. 10805 (2018), 190–209. https://doi.org/10.1007/978-3-319-89960-2_11
[47] The Rust Team. 2020. The Rust programming language. https://rust-lang.org.
[48] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. https://iris-project.org/pdfs/2024-submitted-logical-type-soundness.pdf Manuscript.
[49] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613
[50] Linus Torvalds. 2022. Merge Commit for initial Rust support in the Linux kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b Last accessed 24 Oct 2022.
[51] David Walker. 2005. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press.
[52] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *TASE*. IEEE Computer Society, 44–51. https://doi.org/10.1109/TASE.2018.00014
[53] Qinshi Wang and Andrew W. Appel. 2023. A Solver for Arrays with Concatenation. *JAR* 67, 1 (2023), 4. https://doi.org/10.1007/S10817-022-09654-Y
[54] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *PACMPL* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485522
[55] Sacha Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2024. A hybrid approach to semi-automated Rust verification. arXiv:2403.15122 [cs.PL] https://arxiv.org/abs/2403.15122

Received 2023-11-16; accepted 2024-03-31