Bringing Foundational Verification to Real-World Rust Code

LENNARD GÄHER✉, MPI-SWS, Germany
VINCENT LAFEYCHINE, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, France
SASCHA KEHRLI, MPI-SWS, Germany
AVRAHAM SHINNAR, IBM, USA
WOJCIECH OZGA, IBM Research Zurich, Switzerland
GUERNEY HUNT, IBM Research, USA
DEREK DREYER, MPI-SWS, Germany

Rust is a modern systems programming language that, thanks to its strong memory safety guarantees, is well-suited to the domain of safety-critical systems. Since memory safety alone is not ultimately enough for safety-critical systems, there have emerged in recent years a number of tools for deductive verification of functional correctness of Rust programs. One recent tool, RefinedRust, is notable in that it both handles *unsafe* pointer-manipulating Rust code and produces *foundational*, machine-checked proofs in the Rocq prover. However, RefinedRust is a prototype tool and lacks support for several of the high-level abstractions that Rust provides, including traits, closures, and iterators. These features are commonly used in real-world Rust code, and are supported by other non-foundational Rust verification tools like Prusti and Creusot.

In this paper, we show how to extend RefinedRust with these features, and in a manner such that they can be used *in conjunction with unsafe code*. We demonstrate its usefulness by verifying interesting parts of the memory subsystem of the real-world, low-level ACE security monitor, including its page allocator.

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

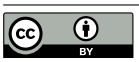
Additional Key Words and Phrases: separation logic, program verification, Rust, Iris

ACM Reference Format:
Lennard Gäher, Vincent Lafeychine, Sascha Kehrli, Avraham Shinnar, Wojciech Ozga, Guerney Hunt, and Derek Dreyer. 2026. Bringing Foundational Verification to Real-World Rust Code. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 352 (October 2026), 27 pages. https://doi.org/10.1145/3839484

1 Introduction

Rust [26] is a modern systems programming language that has seen increasingly widespread adoption in industry as an essential tool for building trustworthy systems code [1, 25]. One of Rust's key selling points is that its core type system guarantees *memory safety*, ruling out common errors made by programmers in legacy systems languages like C and C++, without compromising on performance. This is a primary factor driving its adoption in safety-critical systems like the Linux kernel [27]. For safety-critical programs, however, memory safety alone is not enough: assurances about stronger properties like functional correctness and security are desirable as well.

Authors' Contact Information: Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Vincent Lafeychine, Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France, vincent.lafeychine@ens-paris-saclay.fr; Sascha Kehrli, MPI-SWS, Saarland Informatics Campus, Germany, skehrli@mpi-sws.org; Avraham Shinnar, IBM, Yorktown Heights, USA, shinnar@us.ibm.com; Wojciech Ozga, IBM Research Zurich, Zürich, Switzerland, woz@zurich.ibm.com; Guerney Hunt, IBM Research, Yorktown Heights, USA, gdhhunt@protonmail.com; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.



ACM 2475-1421/2026/10-ART352
https://doi.org/10.1145/3839484

```
pub fn divide_page(mut p: Page) → Vec<Page> {
    let smaller_page_size = p.sz.smaller().unwrap_or(p.sz);
    let number_of_smaller_pages = p.sz.in_bytes() / smaller_page_size.in_bytes();
    (0..number_of_smaller_pages).map(|i: usize| {
      let smaller_page_start = p.address + i * smaller_page_size.in_bytes();
      unsafe { Page::init(smaller_page_start, smaller_page_size) }
    }).collect()
}
```

Fig. 1. Simplified version of page token division in ACE.

Consequently, there has been a great deal of work in recent years on *formal verification* tools for Rust programs [15, 2, 8, 19, 9, 10, 18, 3]. In the interest of building a tool for verification of high-assurance systems code written in Rust, we identify three key desiderata that such a tool would ideally satisfy:

- **High assurance**: For verifying safety-critical code, we would like as strong a formal guarantee as possible. The gold standard would be to aim for *foundational* verification—*i.e.*, a tool producing independent proof certificates that can be machine-checked using a general-purpose theorem prover like Rocq, Isabelle, or Lean, thus eliminating the need to trust the verification tool itself.
- **Unsafe code**: To ensure memory safety, the Rust type system is necessarily quite conservative. As a result, real-world Rust code frequently employs judicious use of *unsafe* Rust features, *e.g.*, manipulation of C-style raw pointers. We therefore want a verification tool that can support reasoning about a mixture of safe and unsafe Rust code.
- **Idiomatic Rust code**: Although Rust is most famous for its ownership-and-borrowing type system, several other of its zero-cost abstractions are also used pervasively in real Rust code, notably: *traits* (Rust's version of type classes), as well as *closures* and *iterators*, which themselves make essential use of traits. To verify real-world *idiomatic* Rust code, it is important for a tool to handle these features.

In our work, we aim to build a verification tool for Rust that satisfies all three of the above desiderata. Towards that end, we have chosen to build on top of RefinedRust [10], as it is the only tool that satisfies both of the first two desiderata: foundational verification of actual Rust programs featuring a mix of safe and unsafe code.[1]

RefinedRust does not, however, satisfy the third desideratum. Gäher et al. [10]'s original paper presented a proof-of-concept verifier for Rust based on the Rocq prover and inspired by the RefinedC [24] system for C verification. It used RefinedRust to verify a condensed version of Rust's `Vec` library of dynamically-growable arrays, featuring the most relevant operations and handling the intricacies of its unsafe implementation. But RefinedRust as presented in that work had only limited applicability to real-world Rust code because it did not support traits, closures, or iterators—features that (as mentioned above) are ubiquitous in real Rust. Some other Rust verification tools do support these features (see §7 for details), but with only limited support for the tricky combination of these features with unsafe code.

As an example of how unsafe code mixes in non-trivial ways with closures and iterators, consider the function `divide_page` in Fig. 1, adapted from the memory management mechanism of the ACE security monitor [22]. It operates on `Page`s, a logical representation of a physical memory page,

[1] RustBelt [15] and RustHornBelt [19] support manual verification of (mixed safe and unsafe) code in the λ_{Rust} type system, but there is a big gap between real Rust and λ_{Rust}, and no automated translation has been developed yet.

asserting exclusive ownership of that page. Internally, it consists of a raw pointer to the base address of the page and the page's size (*e.g.*, 4KiB or 2MiB). `divide_page` takes a page `p`, computes the next smaller page size, and divides the page `p` into a vector of pages of the smaller size, consuming `p`.

The division in `divide_page` is implemented by iterating over the indices of the new pages, and mapping each of them to a newly-created page using the `map` combinator, before finally collecting the new pages into a vector. The new pages are created within a closure passed to the `map` combinator: the closure computes the new page's initial address using pointer-offset operations, and then initializes a new smaller page with this base address. This last operation is *unsafe*: initializing a page requires making sure that the new page has exclusive ownership of its memory region, but the Rust type system cannot reason about the effects of splitting ownership across different indices.

Logically, for verifying safety and functional correctness of the unsafe page initialization, we have to split up the ownership of the page's memory region, and distribute it to the individual (lazy) calls to `map`'s closure when consuming the iterator in `collect`. A key challenge is keeping ownership reasoning compositional: after the call to `map`, further combinators (like `collect`) applied to the iterator should not have to worry about the unsafe code in `map`'s closure.

Making RefinedRust more practical. In this paper, we extend RefinedRust to handle a much wider range of features that are used in real-world Rust code like `divide_page`. Specifically, we analyzed the Rust features employed by the ACE security monitor, a real-world, low-level system requiring high-assurance guarantees, and we determined that the key features lacking useful reasoning principles in RefinedRust were *traits*, *closures*, *iterators*, and *recursive types*.

Iterators. Prusti [5] and Creusot [7] have already shown how to handle iterators in Rust in a modular fashion; thus, as a starting point, we take inspiration from their encoding of iterators. However, Prusti and Creusot are both limited to a first-order specification language, whereas RefinedRust works in a separation logic setting. Moreover, neither of them directly supports verification of iterators in conjunction with closures built from unsafe Rust code.[2] Thus, we need to rethink some aspects of their encoding, in particular when it comes to higher-order iterator combinators.

As an example, consider again the `divide_page` function, which invokes the `map` iterator. In the encoding of iterators in Creusot and Prusti, a caller of `map` has to ensure that the precondition of the closure can always be established; this takes the form of an *iterator invariant*. In `divide_page`, however, the closure passed to `map` is *unsafe*, as it requires extra ownership of the backing memory of a page to execute safely. To that end, when we verify this in RefinedRust, the closure has a separation logic precondition, and the iterator invariant additionally has to carry *ownership*: at every step of the iterator, it has to be decided how to split up the ownership in the invariant in order to establish the precondition of the closure.

We show how to encode this requirement in a general way, generalizing previous specifications to the separation logic setting, and giving expressive specifications to standard Rust higher-order iterator combinators like `map`, `fold`, `try_fold`, and `all` that enable clients like `divide_page`.

Traits. Under the hood, iterators and closures are encoded as traits in Rust's type system. To handle advanced iterator use cases like `divide_page`, we need solid underlying support for traits. Handling traits in RefinedRust comes with its own challenges: it required us to rethink how generic functions are encoded in RefinedRust's semantic type system. A particular challenge, as we will explain in §5, was the interaction of Rust's monomorphization semantics with its elaborate type system. We present a general semantic encoding of traits and trait assumptions supporting

[2] Creusot would allow a function like `divide_page` to be verified, but only by rewriting it using "ghost types" that track memory permissions. The correspondence of this rewritten code to the original code is not always clear. See §7 for details.

expressive functional specifications, which involved non-trivial changes to the function call contract in RefinedRust's type system. More generally, our encoding shows how RustBelt-style semantic models for Rust [15, 19, 6] can be extended to support traits.

Evaluation. As a baseline to show that RefinedRust's support for iterators and closures is sensible, we port the iterator clients from the paper introducing iterators in Creusot [7] to RefinedRust, establishing that RefinedRust can handle common use cases of iterators in safe Rust code.

Moreover, we evaluate the usability of RefinedRust for systems programming by verifying significant parts of the memory management subsystem of IBM's recent ACE security monitor [22]. ACE (Assured Confidential Execution) provides a framework for virtual machine (VM)-based confidential computing in compliance with the RISC-V CoVE specification [4]. ACE enforces isolation between confidential VMs and the hypervisor, providing a trusted execution environment (TEE) where the hypervisor does not need to be trusted. At ACE's core is its *security monitor*, a piece of low-level firmware that configures hardware components to enforce isolation of confidential VMs and provides interfaces for communication between VMs and the hypervisor.

While ACE's security monitor is implemented mostly in safe Rust, crucial parts of the memory management mechanism leverage unsafe Rust. ACE's implementation also makes interesting use of the new features of RefinedRust: it employs recursive types for some of its main datastructures (*e.g.*, its allocator maintains available memory regions using a tree) and it makes heavy use of Rust's high-level programming patterns using iterators and closures, as seen in `divide_page` (Fig. 1). We verify interesting parts of the memory management mechanism of ACE in RefinedRust, including its page token abstraction and its page allocator implementation (around 600 lines of Rust code).

Contributions. We present the following contributions:

- We add support for specifying and verifying traits, closures, and iterators to RefinedRust, extending its foundational type system in Rocq and its Rust frontend.
- Within RefinedRust, we develop new separation logic specifications for Rust's iterators supporting unsafe code, including `map` (§2).
- Using RefinedRust, we verify significant parts of the memory management mechanism of the real-world, low-level, open-source system called ACE (§3).
- We show how our extension of RefinedRust tackles important proof engineering issues in the context of verifying real programs in the Rocq prover (§4).
- We give an account of the semantic encoding of traits in RefinedRust's type system (§5).
- We evaluate our extension of RefinedRust on a variety of case studies (§6).

Limitations and future work. Our work significantly expands RefinedRust's expressive power and brings its capabilities closer to non-foundational verification tools for Rust. Nevertheless, work on improving RefinedRust is still ongoing. The support for specifying closures conveniently could be improved much further, for instance by taking inspiration from Creusot [8], which can infer specifications for simple closures automatically.

The main missing feature requiring conceptual thought is proper support for unsized types, which would enable a cleaner way of supporting slices. This would require non-trivial changes to RefinedRust's semantic model of types, which currently assumes that all types have a finite size.

Some more advanced (and less frequently used) features of Rust, like trait objects and async Rust, are, to the best of our knowledge, not supported by any Rust verification tool at present, and would thus constitute interesting extensions of RefinedRust.

2 Traits and Iterators in RefinedRust

We start with a high-level overview of two of the most interesting new features of RefinedRust — traits and iterators. Iterators are pervasive in real-world Rust programs as they underpin Rust's `for` loops and are thus an essential part of the language. Rust iterators are defined using traits, and iterator combinators are implemented as higher-order functions with closures. The original version of RefinedRust did not support traits or iterators, and thus not even the simplest form of `for` loops.

We use a series of examples of increasing complexity to showcase RefinedRust's new abilities. We start by giving an introduction to RefinedRust's verification style in §2.1. We present a seemingly simple example using `for` loops that nonetheless relies on non-trivial iterators not handled by the original version of RefinedRust. Then, we demonstrate RefinedRust's new ability to verify functions using traits in §2.2. Finally, we show how RefinedRust can now handle the `Iterator` trait (§2.3) and the higher-order iterator adapter `map` (§2.4), key parts of Rust's standard library.

2.1 Introduction to RefinedRust

RefinedRust verifies the functional correctness of Rust programs semi-automatically and modularly, which means that individual functions are given a specification and verified independently of each other to then compose to a verification result for the whole program. To achieve this, the user of RefinedRust specifies the functional behavior of individual Rust functions by annotating them with pre- and postconditions, which relate the *mathematical values* of arguments and return values to each other using assertions in the logic of the Rocq prover. To verify these specifications, RefinedRust employs a *refinement type system*, which extends Rust's type system by adding refinements (*i.e.*, all types come with a notion of *mathematical values* that they are *refined by*).

As an example, consider the function `max_list_i32` in Fig. 2. This function operates on (linked) `List`s of 32-bit signed integers. It takes a *shared* reference `v` of type `&List<i32>` to the list, giving it read access to the list. Then, `max_list_i32` computes the maximum element of the list, or `i32::MIN`, the minimum 32-bit signed integer, in case the list is empty. The implementation itself is straightforward: a mutable variable `m` is initialized to the default value, before using a `for` loop to iteratively compute the maximum. Internally, the `for` loop uses Rust's *iterators* (explained in §2.3).

To verify this function in RefinedRust, we start by giving it a specification in the form of pre- and postconditions. For each of the arguments of the Rust function, RefinedRust assumes a mathematical value that refines the argument and which can be used in pre- and postconditions. Here, references to lists of type `List<i32>` are refined by lists of Rocq integers `list Z`. For `max_list_i32`, we do not need to require any assumptions to verify it, *i.e.*, its precondition is trivial. Thus, we only need to define the postcondition, specifying in line 2 that the list's maximum value has been correctly computed. RefinedRust's notion of correctness is that the function computes a particular mathematical value stated in Rocq, in this case given by the **`rr::returns`** attribute. Here, the computed value is the maximum of the mathematical list `v`, as specified by the function `Z_max_list` (which is defined in the standard way in Rocq as a recursive function). Moreover, RefinedRust always verifies that the function does not cause UB or panics (*e.g.*, that no integer overflows occur).

The original version of RefinedRust was not able to verify this function, as the `for` loop is implemented with the `Iterator` trait. Our new version of RefinedRust enables the user to annotate a *loop invariant* (line 6) on the `for` loop. This example loop invariant asserts that the variable `m` contains the maximum of the elements iterated over so far. It uses the special **`Hist`** variable that RefinedRust makes available in the loop invariant of `for` loops, describing the list of elements previously emitted by the loop's iterator (similar to Denis and Jourdan [7]).[3]

[3] **`Hist`** does not have to correspond to a single specific Rust program variable. If `x` would be an iterator chain instead, **`Hist`** would refer to the history of the whole chain.

```
1  #[rr::returns("
2    Z_max_list (min_int i32) v")]
3  fn max_list_i32(v: &List<i32>) → i32 {
4    let mut m = i32::MIN;
5    #[rr::inv("m =
6      Z_max_list (min_int i32) {Hist}")]
7    for x in v { m = m.max(*x); }
8    m
9  }
```

```
9  #[rr::returns("
10   cmp_max_list {T::O} {T::MIN} v")]
11 fn max_list<T>(v: &List<T>) → T
12 where T: Ord + Min + Copy {
13   let mut m = T::min();
14   #[rr::inv("m =
15     cmp_max_list {T::O} {T::MIN} {Hist}")]
16   for x in v { m = m.max(*x); }
17   m
18 }
```

Fig. 2. Computing the maximum of a list with a RefinedRust specification and loop invariant, for lists over 32-bit signed integers and for generic lists.

Now, to finally verify the specification for `max_list_i32`, we call `cargo refinedrust`, telling RefinedRust to translate the Rust source code to a representation in RefinedRust's operational semantics in Rocq, encode the annotated specification as a Hoare triple, and generate a proof template using the annotated loop invariant. The proof template calls RefinedRust's symbolic execution engine, which in this case proves the specification but leaves one side condition to be manually solved by the user (for re-establishing the loop invariant). Solving this side condition just requires adding a two-line manual Rocq proof to the proof template.

2.2 Introduction to Traits

`max_list_i32` only works for 32-bit signed integers. Ideally, we would generalize it to accept any totally ordered type with a minimal element. Rust supports this using generics and traits, Rust's take on interfaces.

Fig. 2 (right) shows the signature and specification of a generic version of the maximum function. The `where T : ...` clause (line 12) enables the function to be used with any type `T` implementing the `Copy`, `Ord`, and `Min` traits. We explain these requirements before getting to `max_list`'s specification. `Copy` is Rust's built-in trait marking a type whose elements can be copied (*e.g.*, `i32`). These are not subject to Rust's linearity requirements. The `Ord` standard library trait (declared in Fig. 3) marks types which are ordered. Implementors `Self` of the `Ord` trait provide a function `cmp : &Self → &Self → Ordering` (line 10) determining how any two elements of `T` are ordered. Rust's informal standard library contract suggests that this induce a total order for functional correctness.[4] Finally, the `Min` trait is a custom trait we implement on line 15 as an extension of `Ord`, specifying a minimum element of the type according to `cmp`.

In RefinedRust, we can provide and verify specifications for traits and functions that use them. Let us first focus on how we specify traits, before circling back to the specification of `max_list`. Traits can have *specification attributes*, declared using the **`rr::exists`** annotation, which are *logical* components of the trait that implementors have to provide. Conceptually, this approach is similar to how specifications are given to traits in Prusti [5] and Creusot [7], except that those systems distinguish between specification components and proof components, whereas we do not need to because, in Rocq, proof objects are first-class citizens.

The Ord trait. Let us start by explaining (a simplified version of) the `Ord` trait and its specification in more detail. Essentially, we want to formalize Rust's informal contract for `Ord`. First, we provide

[4]But, as with all traits not marked explicitly as `unsafe`, implementors of `unsafe` code may not rely on this functional correctness contract for their safety.

```
1  #[rr::exists("O" :
2    "{Self} → {Self} → order")]
3  #[rr::exists("O_trans" : "Transitive {O}")]
4  #[rr::exists("O_refl" : "Reflexive {O}")]
5  #[rr::exists("O_eq_correct" :
6    "∀ v1 v2, {O} v1 v2 = Equal ↔ v1 = v2")]
7  #[rr::exists("O_antisym" : "...")]
8  pub trait Ord {
9    #[rr::returns("{O} self other")]
10   fn cmp(&self, other: &Self) → Ordering;
11 }
12 #[rr::exists("MIN" : "{Self}")]
13 #[rr::exists("MIN_minimal" :
14   "∀ x, {O} {MIN} x ≠ Greater")]
15 trait Min : Ord {
16   #[rr::returns("{MIN}")] fn min() → Self; }
```

```
17 #[rr::exists("Inv" : "{Self} → iProp")]
18 #[rr::exists("Next" : "{Self} →
19   (option {Item}) → {Self} → iProp")]
20 pub trait Iterator {
21   type Item;
22   #[rr::requires(#iris "
23     {Inv} self.current")]
24   #[rr::exists("s2")]
25   #[rr::observe("self.borrow": "s2")]
26   #[rr::ensures(#iris
27     "{Next} self.current ret s2")]
28   #[rr::ensures(#iris "{Inv} s2")]
29   fn next(&mut self) →
     Option<Self::Item>;
30 }
```

Fig. 3. The `Ord` (simplified) and `Min` traits (left) and the `Iterator` trait (simplified, right).

an attribute `O` (line 2) specifying a *mathematical function* (in Rocq), mirroring the Rust `cmp` function. This implies that comparison has to be deterministic. Secondly, the attribute `O_trans` (line 3) is a proof that `O` is transitive, and similarly, `O_refl` and `O_antisym` require reflexivity and antisymmetry, respectively. Finally, the attribute `O_eq_correct` (line 6) is a proof that `O` identifies two elements as equal exactly when they are equal according to Rocq's notion of equality. Note that attributes are allowed to depend on each other (used here to specify constraints on the `O` relation).

Apart from declaring logic-level attributes, a specification for a trait must also provide default specifications for all its methods, which we call *base specifications*. Every implementation of the trait verified by RefinedRust is required to satisfy the base specifications of its methods, but may optionally override those with more precise, instance-specific specs so long as they imply the base specs. As a result, code that operates over a generic type `T` satisfying the trait can assume that `T`'s implementation of the trait at least satisfies the base specs, as long as clients of this code are also verified with RefinedRust. In the case of `Ord`, we have to provide a specification for `cmp`. We define that the result should reflect the logic-level function `O`, *i.e.*, `cmp` should correctly implement `O`.

The Min trait. The specification for `Min` declares the existence of a minimal mathematical value `MIN` according to the order `O`, which is returned by the `min` function.

Specifying max_list. We are now ready to give a specification to `max_list`, the generic list maximum function (Fig. 2). The core specification is the same as for `max_list_i32`, but uses `Ord`'s order instead of the implicit order on integers. To enable this, the specification can mention all specification attributes declared on the traits that have been required in `where` clauses (here, the order `O` and the minimum `MIN`). The return value is specified by a Rocq function `cmp_max_list`, which is similarly parametric in the order.

2.3 Iterators

In the above examples, we were iterating over the `List` using `for` loops. Rust `for` loops are implemented using the `Iterator` trait, providing a generic abstraction for iterating over sequences of elements. The `List` type, for example, implements the `Iterator` trait to enable iteration over its elements. Since iteration is such a fundamental abstraction, the `Iterator` trait is one of the most

commonly used traits in Rust. It is also challenging to formally specify in a sufficiently general way: iterators are implemented for a wide variety of data structures, and feature higher-order iterator combinators like `map` or `filter`. In the following, we give a broad overview of how iterators are specified in RefinedRust. Our specifications adapt insights from existing specifications in Creusot [7] and Prusti [5] to RefinedRust's separation logic setting (which allows it to seamlessly handle ownership reasoning not understood by the Rust type system).

The Iterator trait. Fig. 3 (right) shows the declaration of the iterator trait along with its specification. Let us focus on the Rust declaration without the specification. The iterator trait, at its core, has two components: the associated type `Item` and the method `next`. The `Item` type specifies the type of elements the iterator emits. The `next` method takes a mutable reference to the iterator state and optionally returns the next element the iterator can emit, while advancing the state.

In order to describe the intended behavior of implementations of the `Iterator` trait, we first annotate the trait declaration with the additional specification attributes `Next` (line 19), giving a relational specification of the evolution of the iterator state, and `Inv` (line 17). `Inv` is a predicate that can capture invariants for functional correctness specific to a particular iterator. For any `s1`, `e` and `s2`, the assertion `Next s1 e s2` states that in the iterator state `s1`, the iterator can emit the optional element `e` and then transition to the iterator's state `s2`. Note that this gives a functional perspective on the evolution of the iterator's state: we have simultaneous access to the current state `s1` and the next state `s2`, which allows us to relate them.

`Next` is declared to be a predicate of type `iProp`, using RefinedRust's language of *separation-logic propositions* that we inherited from the Iris separation logic framework. In contrast to Rocq's language of propositions `Prop`, this allows it to capture additional ownership (for instance, of memory not automatically managed by Rust's type system) using Iris separation logic assertions; this will be useful when verifying `divide_page` from Fig. 1 in §3.1.

We use `Next` to specify the `next` method. First, let us explain how the mutable reference to `self` is handled in RefinedRust. The mathematical value of a mutable reference is a tuple, consisting of the current value stored at the mutable reference, `self.current`, and a RefinedRust *borrow variable* `self.borrow` used for transmitting information about how the mutable reference is updated to its lender (in this case, at the call site of `next`). Once the mutable reference goes out of scope, RefinedRust generates an *observation* on `self.borrow`, assigning it the final value stored at the mutable reference, which can then be used by the lender to obtain the updated value.

Now, first, calling `next` requires that the iterator's invariant `Inv` is upheld (line 23). Here, we use the **`#iris`** modifier, stating that the following proposition is an Iris separation-logic proposition. Then, we declare that `next` returns an optional item **`ret`** (implicitly quantified), while evolving the iterator state from `self.current` to `s2` (line 25). The **`rr::observe`** clause states an observation on the evolution of the mutable reference to `self`, providing the new value of `self` after the function returns. This transition needs to obey the `Next` relation (line 27), and, moreover, `Inv` needs to be satisfied in the new state `s2` (line 28). Note that, since `Next` is a separation logic predicate, this enables an iterator to emit arbitrary separation logic ownership in each step.

2.4 Higher-Order Iterator Specification

A crucial part of iterators are *iterator adapters*, which allow transforming the sequence of elements emitted by an iterator. One example of this is the `map` adapter used in the `divide_page` method of Fig. 1, which maps a closure over every element of the iterator. This closure may be stateful, modifying its captured state, which already makes specifications for `map` in Prusti [5] and Creusot [7] highly non-trivial. In addition, in RefinedRust, the closure *may perform unsafe operations* with state that is not tracked by Rust, as seen in Fig. 1. This statefulness makes `map` challenging to specify and

```
1  pub struct Map<I, F> { it: I, f: F, }
2  #[rr::instantiate("Inv" := "λ (it, f),
3  ∃ MapInv, {I::Inv} it ∗ MapInv it f ∗ ...")]
4  #[rr::instantiate("Next" := "λ s1 e s2,
5  if_None e ({I::Next} s1.(it) None s2.(it)) ∗
6  if_Some e (λ e, ∃ e_inner,
7    {I::Next} s1.(it) (Some e_inner) s2.(it) ∗
8    OnlyPersistent ({F::Pre} s1.(f) e_inner) ∗
9    {F::Post} s1.(f) e_inner s2.(f) e)")]
10 impl<B, I: Iterator, F> Iterator
11   for Map<I, B, F>
12   where F: FnMut(I::Item) → B {
13   type Item = B;
14   fn next(&mut self) → Option<B>
15   { self.it.next().map(&mut self.f) }
16 }
17 #[rr::params("MapInv" : "
18   {I} → {F} → iProp")]
19 #[rr::requires(#iris "MapInv it f")]
20 #[rr::requires(#iris "{I::Inv} it")]
21 #[rr::requires(#iris "□ (∀ it it' f e,
22 OnlyPersistent (
23   {Self::Next} it (Some e) it') -∗
24   Inv it f -∗ ({F::Pre} f e ∗
25   (∀ e' f', OnlyPersistent (
26     {F::Post} f e f' e') -∗
27     Inv it' f')))")]
28 #[rr::requires(#iris "□(∀ it ...")]
29 #[rr::ensures(#iris "{Map::Inv} (it, f)")]
30 #[rr::returns("(it, f)")]
31 fn map<I: Iterator, B, F>(it: I, f: F)
32   → Map<I, F> where F: FnMut(I::Item) → B
33 { Map { it, f } }
```

Fig. 4. `Iterator` implementation for `Map`.

verify, as we need to account for arbitrary state modifications in a potentially *unsafe* way that is not tracked by Rust's type system, and necessitates *separation logic* specifications.

Let us start by explaining the implementation of `map` (line 31 in Fig. 4). The `map` function takes an existing iterator `it` and a closure `f` and returns a new instance of the `Map` structure—essentially a tuple of the inner iterator `it` and the closure `f`. The type `F` of the closure `f` is required to implement the `FnMut` trait for arguments of type `I::Item` and some return type `B`, allowing the closure to mutate the closure state (*e.g.*, captured variables). Note that `map` does not apply `f` to anything yet, as Rust's iterators are lazy and only yield elements once their `next` function is called.

The interesting part of `Map` is its `Iterator` implementation. The `Item` type (line 13) is declared to match the return type `B` of the closure. Then, the `next` implementation (line 15) queries the inner iterator for the next element, and uses `map` on `Option<I::Item>` to optionally apply the closure to the next element.

Now, we need to prove that `Map` actually satisfies the `Iterator` specification (Fig. 3), ensuring that `next` satisfies the base specification. First, calling the inner iterator's `next` function only requires the nested iterator's invariant. Then, we need to distinguish two cases: either `next` returns `None`, in which case we are done, or `next` returns some element, in which case we apply `f` to it. However, the closure `f` may have an arbitrary precondition! (We will see an example of this when verifying the code from Fig. 1 in §3.1.) RefinedRust needs to establish this precondition when calling `f`. The most straightforward way to ensure that this precondition is satisfied would be to require the precondition of `f` in the precondition of `next`. However, this would leak the specification of the closure to the outside, requiring the consumer of the iterator to reason about the composition of the iterator.

The key to solving this problem is to equip `Map` with an *invariant* that is strong enough to call `f` in every `next` call of the iterator. Our specification will allow the caller of `map` to pick a separation-logic invariant `MapInv` for `Map` that depends on the current iterator and closure state.

Another key question is how to split up the ownership that is emitted by the postcondition of the closure `f` and the `Next` relation of the inner iterator `it`: since separation logic propositions are not duplicable in general, we may either use it to re-establish the invariant `MapInv` after the closure

returns, or emit it to the consumer of the `Map` iterator via its `Next` relation, but not both. For our specification, we take the choice to always emit this ownership to the consumer of the iterator.[5]

We still allow persistent (in particular, duplicable) parts of this ownership to be used when re-establishing `MapInv`. For this, we use RefinedRust's `OnlyPersistent` modality, which slices out the persistent (thus, duplicable) parts of a proposition.

Specifying map. To make matters concrete, let us look at the specification for the `map` function. It assumes the existence of the invariant `MapInv` in line 18. Moreover, the invariant always needs to hold of the current iterator and closure state (line 19). In addition, the invariant of the nested iterator needs to hold (line 20).

The most crucial property (from line 21) is that the invariant is strong enough to prove the closure's precondition, and using the closure's postcondition, re-restablish the invariant. Since all of these components are separation logic assertions, this property is stated using separation logic connectives, in particular the magic wand $-\!\!*$, separation logic's ownership-carrying version of implication, stating that ownership of one resource can be used to obtain ownership of another resource. Now, given that the inner iterator has emitted a new element (line 23) and that the invariant currently holds (line 24), we can prove the precondition of the closure dependent on the closure state and the argument (line 24). Here, we wrap the `Next` relation of the inner iterator in `OnlyPersistent` as the contained ownership will be emitted to the consumer of the iterator. Then, for any return value and new closure state, the postcondition of the closure (depending on the old closure state, argument, new closure state, and return value) is sufficient to establish the invariant again for the new states (line 27). Note that all of this needs to hold *persistently* as indicated by Iris's "persistently" modality $\square$, *i.e.*, it can be used multiple times. A similar invariant also needs to hold for the case that the inner iterator emits no element (omitted, line 28). Finally, `map` ensures that the invariant of the `Map` iterator (to be defined below) is correctly initialized (line 29).

Specifying Map. At long last, we prove that `Map` correctly instantiates the `Iterator` trait. First, we instantiate the `Inv` attribute (line 3), by declaring the existence of `MapInv`, satisfying the various properties from the precondition of `map`. For the `Next` relation, there are two cases. If the iterator does not emit an element (`e` is `None`), then the inner iterator also emits no element (line 5). If the iterator emits an element, then there needs to be an element e_{inner} emitted by the inner iterator (line 6) which satisfies its `Next` relation (line 7). The user of `next` also obtains the knowledge that e_{inner} satisfies the closure's precondition (line 8, under a `OnlyPersistent` modality), and that according to the closure's postcondition, e is a valid return value of the closure (line 9).

3 Case Study: Page Allocation in the ACE Security Monitor

To demonstrate the improved scalability and capabilities of RefinedRust, we verify security-critical components of a real-world, low-level firmware, the ACE security monitor [22]. This firmware is part of the ACE (Assured Confidential Execution) framework, an open-source virtual machine (VM)-based confidential computing technology for RISC-V systems. The ACE security monitor's main responsibility is to enforce isolation between different security domains (*i.e.*, VMs and a hypervisor), using software mechanisms and underlying hardware support.

The ACE security monitor (SM) adds to virtualization's partitioning of hardware in space and time, giving each security domain an impression that it executes on an isolated computing environment. To enforce this level of isolation, the SM maintains control of physical memory allocation and context switches, *i.e.*, changes of control flow between security domains. During a context switch,

[5]While it is possible to prove a more general specification leaving this choice to the client of `map`, such a specification is harder to automate reasonably.

```
1  #[rr::refined_by("p" : "page")]
2  #[rr::invariant(#type "p.(page_loc)" : "<#> p.(page_val)" @
3    "array usize (page_size_in_words p.(page_sz))")]
4  #[rr::invariant("(page_end_loc p) ≤ MAX_PAGE_ADDR")]
5  #[rr::invariant("page_wf p")] #[rr::invariant("...")]
6  pub struct Page {
7  #[rr::field("p.(page_loc)")] addr: ConfidentialMemoryAddress,
8  #[rr::field("p.(page_sz)")] size: PageSize, }
9
10 #[rr::requires(#type "addr" : "..." @ "array ...")]
11 pub unsafe fn init(addr: ConfidentialMemoryAddress, size: PageSize) → Page { ... }
```

Fig. 5. Simplified Rust definition of page tokens.

the SM reconfigures hardware to enforce memory access protection and clears the architectural and micro-architectural state containing traces of a security domain's execution. The resumed security domain execution is then constrained by the hardware configuration: for instance, the memory management unit (MMU) controls the VM's address translation, and thus which regions of physical memory a VM can access. We focus our efforts on verifying the core abstractions inside ACE that the MMU configuration relies on.

ACE memory layout. During system initialization, ACE splits the machine's physical memory into memory managed by the hypervisor (non-confidential memory) and memory managed by the security monitor (confidential memory). These two memory regions are disjoint and the split is hardware-enforced for the system's entire lifetime. To ensure isolation between hypervisor and VMs, the security monitor relies on a global `MemoryLayout` configuration storing the memory boundaries of confidential and non-confidential memory, thus allowing it to distinguish between them. The `MemoryLayout` remains constant after system initialization.

The SM sanitizes addresses originating from a VM or a hypervisor by checking that they belong to the expected (confidential or non-confidential) memory region. To do so, the SM represents memory addresses using dedicated types like `ConfidentialMemoryAddress`, which wrap Rust's raw pointers. Using `MemoryLayout`, the SM ensures that offset operations on `ConfidentialMemoryAddress` stay in the confidential region.

Key abstraction: Page tokens. Let us now focus on how the security monitor assigns confidential memory to VMs and enforces VM isolation. The SM uses the memory management unit (MMU) hardware to assign memory to VMs at page-level granularity and to enforce that a VM accesses only memory it owns. To correctly assign confidential memory to VMs, the SM must manage memory in chunks that are aligned with RISC-V MMU-defined page boundaries.

ACE introduces *page tokens* as the core abstraction for handling chunks of page-aligned memory (Fig. 5). The key components are the `address` of the page's start and the `size` of the page (*e.g.*, `4KiB`, `2MiB`, `1GiB`, ...). Note that page tokens have to reside fully in confidential memory.

Our RefinedRust specification of page tokens is annotated on the structure. First of all, we declare a mathematical model `page` for page tokens, containing: the memory location `page_loc`, the stored sequence of words `page_val`, and the size `page_sz`. Besides several functional properties, the most important part of the invariant is a *RefinedRust type assignment* for the backing memory of the page: using RefinedRust's **`#type`** modifier, the invariant in line 2 states that `p.(page_loc)` has the RefinedRust `array` type, exclusively owning it and containing the list of words `p.(page_val)`.

```
1  #[rr::ensures("subdivided_pages self ret")]
2  pub fn divide(mut self) → Vec<Page> {
3    let smaller_sz = self.size.smaller()
4      .unwrap_or(self.size);
5    let num_pages = self.size.in_bytes() /
6      smaller_sz.in_bytes();
7    let page_end = self.end_address_ptr();
8    let memory_layout = MemoryLayout::read();
9    (0..num_pages).map(|i: usize| {
10     let offset_in_bytes = i * smaller_sz.in_bytes();
11     let smaller_page_start =
12       memory_layout.confidential_address_at_offset(
13         &self.addr, offset_in_bytes, page_end)
14       .unwrap();
15     unsafe { Page::init(smaller_page_start, smaller_sz) }
16   }).collect()
17 }
```

```
18 #[rr::params("v")]
19 #[rr::requires("page_end ≤
20   memory_layout.(conf_end)")]
21 #[rr::requires("{self.addr} +
22   (S i) * smaller_sz ≤ page_end")]
23 #[rr::requires(#type
24 "{self.addr} +ₗ (i * smaller_size)" :
25 "<#> v" @ "array usize smaller_sz")]
26 #[rr::returns("
27   mk_page ({self.addr} +ₗ
28   (i * smaller_sz)) smaller_sz v")]
29 |i: usize| { ... }
```

Fig. 6. Division of page tokens (left) and specification for the closure passed to `map` (right).

It is `unsafe` to create a page token with the constructor `init` (line 11): page tokens assert exclusive ownership over their region of memory and this ownership is not tracked by the Rust type system; it is a precondition the caller needs to ensure. RefinedRust makes this precise by requiring the caller to provide ownership of the backing memory (line 10) much like in the page token invariant.

Page allocator. During system initialization, the security monitor instantiates page tokens by logically dividing the confidential memory region into pages. This task is handled by the page allocator, a dedicated component responsible for managing page allocation. Throughout system execution, other components of the SM can invoke it to request page allocation and deallocation.

To efficiently manage available page tokens, the page allocator prioritizes maintaining the largest possible pages and dynamically splits them into smaller ones only when necessary. Thus, two key operations on page tokens are `divide` (page token division) and `merge` (page token merge). We focus now on the implementation `divide` because it shows a particularly interesting use case of iterators.

3.1 Verifying Unsafe Higher-Order Iterators

Fig. 6 shows the implementation of `divide`: The `self` page token is consumed, and smaller page tokens are created in its place. In Rust's ownership type system, this is expressed by taking full ownership of `self` and returning ownership of a vector of new page tokens. Our RefinedRust specification functionally specifies the returned pages with the `subdivided_pages` relation in Rocq.

The Rust implementation proceeds in multiple steps: (1) Determine the next smaller page size (line 4). (2) Calculate the number of smaller pages to create (line 6). (3) Iterate over the indices of smaller pages (line 9) using an iterator. For each smaller page, the implementation computes its offset (line 10) and its starting address (line 11), and finally creates a page token at this offset (line 15).

The use of iterators in the last step is a particularly challenging one, as we pass a closure encapsulating *unsafe* code to the `map` adapter. In particular, the closure creates a new (smaller) page token at the index it receives as argument, using the `Page::init` method shown before, and returns

it. For creating this page token, we need to split up the ownership of the backing memory of the original page into smaller chunks and then pass this extra piece of ownership into the closure.[6]

In order to prove the ownership transfer correct, we use the expressive specifications for iterators and the `map` iterator adapter shown in §2.4. To do the verification, we need to perform three key steps: (1) First, before the creation of the iterator chain (which instantiates new page tokens), we have to split up the ownership of the original page token's memory. This is the key part of the proof that happens manually, and requires human insight to perform a few manual proof steps (explained below). (2) Secondly, we need to give a specification to the closure passed to map. (3) Finally, we need to define an invariant on the `map` adapter which specifies how the ownership of the page token fragments is dispensed on each application of the closure.

Specification. We start by specifying the closure passed to `map` in Fig. 6. Closures in Rust may capture parts of their environment — for instance, in this case, the closure uses the `memory_layout` variable and `self.addr` from the surrounding function. Captures are computed automatically by the Rust compiler, and the specification may mention captured places from the surrounding function scope. First, we require that the page to be created will reside within the upper bounds of the old page (line 22). Moreover, we require ownership of the memory region of the new page, using the array type (line 25). Finally, the closure returns a page of the smaller size (line 28), starting at a positive offset from `self.addr`, using the mathematical offset operation $+_l$.[7]

Invariant. Now, `map` needs to prove the preconditions required by the closure for each element of the iterator; in particular it has to provide the ownership of the backing memory of each page. Here, crucially, we finally reap the benefits of specifying `Map` to support separation-logic invariants in §2.4! In our case, we pick as the invariant the ownership needed for the remaining iterations, which depends on the current iterator state s (`it_state`), the smaller page size sz' (`smaller_size`), and the page value *page* (`self.page_val`), stated using Iris's iterated separating conjunction:

$$\ast_{i \in s.\texttt{start} \ldots s.\texttt{end}} \; l +_l (i * sz') : \quad page[sz' * i \ldots sz' * (1 + i)]@\texttt{array}\ (\texttt{usize})\ sz'$$

Proof. The proof requires manual proof steps which are not automated by RefinedRust, in particular for (1) specifying the invariant above for `map`, (2) showing that the invariant satisfies `map`'s preservation conditions, and (3) showing that the vector of `collect`ed pages together satisfies the postcondition of the function. Step (1) requires stating the invariant above in Rocq. For step (2), we need to reason about splitting up the array ownership, which RefinedRust does not currently automate. For step (3), RefinedRust automatically derives an inductive property about the iteration sequence, but we need to manually do an induction in order to show that this implies the `subdivided_pages` predicate.

4 Implementation

Our extension of RefinedRust introduces several enhancements to support broader adoption, with a particular focus on improving usability and streamlining the proof development workflow. Below, we highlight selected modifications that are particularly relevant from a proof engineering perspective.

[6]Note that Rust does not have a notion of unsafe closures. However, morally, this closure should be marked as unsafe, as it is not always safe to call.

[7]Note that the specification specifies implicitly (through the page token invariant) that all of the memory required in the precondition is consumed for the new page.

4.1 Simplified Specification Language

Specification languages generally have to relate the mathematical world (in which the specification is written) and the physical world (in which the program is written). As distinguishing the two worlds is a significant source of overhead when writing specifications, well-engineered verification tools frequently try to conflate the two as much as possible (so much so that Rust verification tools like Creusot [8], Prusti [2], or Verus [18] use Rust itself as a specification language). Our extension of RefinedRust follows this approach, and the specifications shown in this paper conflate mathematical variables and program variables by design.

In contrast, the original version of RefinedRust (following RefinedC [24]) enforced a strict separation between mathematical variables and program variables, resulting in unnecessarily verbose specifications and reduced usability, particularly in large-scale verification efforts. For example, the `Iterator::next` specification (Fig. 3) would have looked as follows in the original version of RefinedRust:

```
#[rr::params("it", "γ") #[rr::args("(#it, γ)")]
#[rr::requires(#iris "{Inv} it")]
#[rr::exists("it'", "elem")]
#[rr::ensures(#iris "{Next} it elem it')] #[rr::ensures(#iris "{Inv} it'")]
#[rr::returns("<#> elem")] #[rr::observe("γ": "it'")]
fn next(&mut self) → Option<Self::Item>;
```

Here, **`rr::params`** explicitly quantifies universally-quantified Rocq-level parameters, which are then related to the program variables with **`rr::args`** and **`rr::returns`**. Note the injections # here: these are related to RefinedRust's encoding of refinements of *mutable references*, and were a key obstacle for implementing RefinedRust's new specification language.

Specifically, RefinedRust's mathematical model for mutable references takes inspiration from RustHorn's prophecy approach [20], and represents them approximately by a tuple (x, γ), where x is the current value and γ is a so-called *borrow name*. The symbolic borrow name will be used to communicate the final value of the reference when its lifetime ends. In order to make this work, every place that can be mutably borrowed is represented by either a known value (#x) or a symbolic reference to a borrow name ($*\gamma$).

Our key insight is that values crossing a function boundary are always concretely known: if they were mutably borrowed before, the Rust borrow checker enforces that the borrow has ended, and so their final value is known. Thus, we can assume that all function arguments do not contain symbolic references, and restrict to the #x case.

RefinedRust's new specification language implicitly quantifies over borrow-name-free mathematical values for all function arguments. Only when a function returns a *reborrow* of a mutable reference do the injections still have to be specified explicitly for the reborrow.

4.2 Mixing Manual and Automatic Proofs

A key question for semi-automatic verification tools like RefinedRust is how to mix automatic proofs and user-provided proofs. The original version of RefinedRust followed RefinedC's proof paradigm, where the type-checking process is expected to succeed fully automatically, while custom sequences of Rocq tactics for solving side conditions can be annotated in the program source.

For our extension of RefinedRust, we abandon this proof paradigm and develop a new, *more interactive* style of proofs mixing manual and automatic reasoning, believing this to be better suited for verifying Rust programs. Now, RefinedRust creates a *proof template* in Rocq for each function (or other proof obligation), containing the core of the proof. The proof template is *persistent*, and

can be safely modified by the user and version controlled. The proof statement itself and a proof prelude are generated in a separate file and are re-generated on changes to the program source.

Additionally, RefinedRust's type checking loop is now designed for interactivity: When the type system cannot make progress, the proof state is restored to the last *stable state*, which is deterministically placed based on the syntax of the program (e.g. at statements). This enables the proof engineer to manually transform the proof context (*e.g.*, executing ghost state updates or asserting intermediate proof goals), before proceeding with RefinedRust's typechecker. Returning to a *stable state* avoids exposing the proof engineer to details of RefinedRust's internal proof states that have little connection to surface Rust code.

5 Trait Semantics

RefinedRust's frontend encodes Rust's MIR (Mid-level Intermediate Representation) into the so-called Radium operational semantics inside Rocq. The Radium semantics is RefinedRust's source of ground truth about the execution behavior of Rust programs. As such, Radium and the RefinedRust encoding into it are core parts of the TCB (trusted computing base). Radium represents MIR code after type checking, trait resolution, borrow checking, and drop elaboration. This means that the `rustc` compiler first checks and transforms the surface-level Rust source code using these compiler passes before RefinedRust sees it.[8]

Our approach has both pluses and minuses. One downside, from the perspective of verifying surface-level Rust code, is that we must trust that the (non-trivial) transformations implemented in `rustc` match the programmer's mental model of the surface-level Rust semantics. However, pending the development of a formal specification of surface-level Rust semantics, the `rustc` implementation of these transformations *is* the de-facto semantics of Rust. Additionally, the thus-transformed MIR code (which we then verify) has the advantage of being relatively far down the `rustc` compilation pipeline, thus reducing the number of further, lower-level `rustc` transformations that one must trust in order to obtain correctness guarantees about the final compiled assembly code.

Radium is monomorphic. In order to give a semantics to MIR functions pre-monomorphization, RefinedRust parameterizes the translated MIR code by the layout information that `rustc`'s monomorphization pass further down the compilation pipeline computes. Nevertheless, RefinedRust can verify polymorphic specifications against parameterized versions of Radium's monomorphic code (effectively giving a polymorphic semantics), which can still later on be monomorphized to compose to a concrete monomorphic Rust program.

Thus, ultimately, the top-level soundness theorem of the original version of RefinedRust states a soundness result about full programs composed of monomorphized functions:

Theorem 5.1 (Adequacy [10]). *Let $\mathcal{F}$ be a "main" function for which the RefinedRust type system (instantiated with a Rust 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.*

Our extended version of RefinedRust retains this theorem essentially unchanged. However, we need to ensure that our extensions to RefinedRust do not break the ability to instantiate this theorem for concrete programs.

In the rest of this section, we give an overview of our underlying semantics and encoding of traits after RefinedRust has resolved all trait impls. At a high level, our encoding reduces each trait declaration to a set of Rocq record declarations stating components and properties of implementors of this trait, and each trait implementation to an instance of these records. More interesting, as we

[8] This approach is common to virtually all deductive Rust verification tools.

shall see, is how we handle functions with trait bounds on their type parameters (and the callers of those functions): this required fundamentally expanding the caller-callee-contract in order to allow correct linking with particular trait implementations.

5.1 Semantic Encoding of Traits

The RefinedRust frontend encodes all trait declarations and trait implementations as Rocq definitions as part of RefinedRust's semantic type system.

When translating a *trait declaration*, RefinedRust generates two record types that capture the signature of the trait. These record types will later be inhabited by impls of the trait. First, the *spec record type* is a type that specifies the generic signature of the trait's methods. Secondly, the *attribute record type* is a type that specifies the type signature of logic-level specification attributes annotated on the trait (using **`rr::exists`** annotations). In addition, RefinedRust generates a (reflexive & transitive) inclusion relation between different instances of the spec record type, ordering specifications by strength. (Typically, this just lifts function subtyping pointwise.) Finally, RefinedRust generates an *instance* of the spec record type capturing the *base specification* that has been annotated by the user on the Rust trait declaration (*e.g.*, see Fig. 3), parametric in an instance of the attribute record type.

When translating a *trait implementation*, RefinedRust generates *instances* of the spec record type and attribute record type for this particular impl: the spec record instance contains specifications for each of the impl's methods, while the attribute record instance specifies the instantiation of the attributes declared on the trait using **`rr::instantiate`** annotations (see §2). Finally, RefinedRust generates a proof showing that the impl's spec implies the trait's base specification.

When translating an object with *trait requirements*, the RefinedRust frontend elaborates the trait requirements into a semantic representation where all statically inferrable information has been resolved (using the previously encoded trait declarations and trait implementations). For trait resolution, we invoke `rustc`'s queries which are usually performed as part of its codegen pass (*i.e.*, when lowering MIR code into LLVM IR). Doing so ensures that RefinedRust resolves the same trait implementations for the Radium model that `rustc` resolves for the compiled Rust code. Afterwards, all instantiations of trait requirements (*e.g.*, when calling a function or using a trait with trait requirements) have been resolved to either a trait requirement on a type parameter or to a concrete trait implementation.

Example: Encoding the Iterator trait. Now, let us look at the encoding of the `Iterator` trait as a concrete example (shown in Fig. 3). For brevity, we shorten `Iterator` to `Iter` in the following.

Before we explain the encoding, we need some background on how types are encoded in RefinedRust. In RefinedRust, each type consists of several components: the *syntactic type* (determining the Rust data layout of the type), the *mathematical type* (*i.e.*, the mathematical refinement type used in specifications), and the *semantic type* (giving a semantic interpretation of the type in terms of a predicate on program values). For instance, the following RefinedRust types correspond to the Rust type `&i32`: its syntactic type describes the data layout of (thin) pointers, its mathematical type is the Rocq integer type `Z`, and its semantic type is a record describing its semantics in the RefinedRust type system (*i.e.*, its safety and representation invariants) using Iris predicates.

For a type parameter `T`, we use *TSt* to refer to its syntactic type, *T* for its mathematical type, and T for its semantic type (and similarly for `Self` and `Item`).

Returning to the encoding of the iterator trait, RefinedRust first generates the specification record type `Iter_spec` and the attribute record type `Iter_attrs`. First, the `Iter_attrs` record contains the declaration of the user-annotated attributes `Next` and `Inv` on the trait declaration, being parametric over the mathematical types (of type *RT*) of the implementor of the trait (*Self*), the type parameters

of the trait (in this case, none), and the trait's associated types (*Item*).

$$\texttt{Iter\_attrs}(\mathit{Self} : \mathit{RT})(\mathit{Item} : \mathit{RT}) := \{\texttt{Iter\_Next} : \mathit{Self} \to \mathsf{option}\ \mathit{Item} \to \mathit{Self} \to \mathit{iProp};\ \ \texttt{Iter\_Inv} : \ldots\}$$

Next, the specification record `Iter_spec` declares the type signature of each of the trait's methods. In this case, we only give the specification `Iter_next_spec` of the `next` method: its type `generic_spec` $(1, [])$ `fnspec` means that it is a function specification (of type `fnspec`, encoding the function signature and pre- and postconditions for RefinedRust) that has one lifetime parameter (the lifetime of the reference argument `&mut self`) and no type parameters.

$$\texttt{Iter\_spec}(\mathit{Self} : \mathit{RT})(\mathit{Item} : \mathit{RT}) := \{\texttt{Iter\_next\_spec} : \texttt{generic\_spec}\ (1, [])\ \texttt{fnspec}\}$$

Note that the specification record type `Iter_spec` is not parametric in the attributes `Iter_attrs`, but that each concrete instance of `Iter_spec` may well rely on a concrete instance of `Iter_attrs`.

The inclusion relation lifts RefinedRust's notion of function subtyping $\subseteq_{\mathsf{fn}}$ pointwise over all of the trait's function specifications, in this case only for `Iter::next`:

$$\texttt{Iter\_incl}\ \mathit{Self}\ \mathit{Item}(S_1\ S_2 : \texttt{Iter\_spec}\ \mathit{Self}\ \mathit{Item}) := S_1.(\texttt{Iter\_next\_spec}) \subseteq_{\mathsf{fn}} S_2.(\texttt{Iter\_next\_spec})$$

We use the notation $S_1 \subseteq_{\texttt{Iter}} S_2$ to denote `Iter_incl` $S_1\ S_2$, leaving the type parameters implicit.

Finally, the base specification `Iter_base_spec` bundles up all the user-annotated base specifications declared on the trait methods into an inhabitant of the spec record type `Iter_spec`. It is parametric in the generic context of the trait (containing the type parameter `Self` and the associated type `Item`) as well as the attributes *attrs* declared on a potential implementation. It then instantiates the specifications generated for each of the trait's methods. In this case, the base specification just consists of the base specification for `next`, `Iter_next_base_spec` (omitted):

$$\begin{aligned}&\texttt{Iter\_base\_spec}(\mathit{Self}\ \mathit{Item} : \mathit{RT})(\mathit{SelfSt}\ \mathit{ItemSt} : \mathit{SynType})(\mathit{attrs} : \texttt{Iter\_attrs}\ \mathit{Self}\ \mathit{Item}) :\\ &\quad \texttt{generic\_spec}\ 0\ [\mathit{Self}; \mathit{Item}]\ (\texttt{Iter\_spec}\ \mathit{Self}\ \mathit{Item}) :=\\ &\quad \lambda \mathsf{Self}, \mathsf{Item}.\ \{\texttt{Iter\_next\_spec} := \texttt{Iter\_next\_base\_spec}\ \mathit{Self}\ \mathit{Item}\ \mathit{SelfSt}\ \mathit{ItemSt}\ \mathit{attrs}\ \mathsf{Self}\ \mathsf{Item}\}\end{aligned}$$

Here, the special quantifier λSelf, Item binds the lifetime and semantic type parameters—in this case, no lifetimes, but two semantic types for `Self` and `Item` are quantified. (For brevity, we leave the constraint to be compatible with *Self* and *SelfSt* resp. *Item* and *ItemSt* implicit on paper.)

5.2 Caller-Callee Contract with Traits

The most crucial part of the semantics of traits is the caller-callee contract: what can a function with trait requirements assume, and what does its caller have to guarantee? This is particularly interesting for RefinedRust's semantic type system, as it verifies generic functions polymorphically, while the Rust compiler monomorphizes all generic functions during compilation: in other words, RefinedRust has to reason polymorphically about monomorphic functions. To understand better what this entails, let us first explain how RefinedRust handles type and lifetime parameters *in the absence of traits*, before later adding traits to the mix.

Static and dynamic components. In RefinedRust, functions can be generic in lifetime parameters and type parameters. Conceptually, these parameters exhibit a kind of *phase distinction* [14], splitting into *static* and *dynamic* components. We color-code static components in red and dynamic components in blue. Static components are instantiated in the statement of the correctness proof of the function (or, viewed from the Rust compiler's perspective, when monomorphizing code). On the other hand, dynamic components are only instantiated during type checking when calling the function from another function. The syntactic type *WSt* and the mathematical type W are both static components. On the other hand, lifetime parameters `'a` are dynamic, as the lifetime that a function is instantiated with may be local to the function that calls it. Similarly, semantic types W are dynamic, as they may depend on lifetimes (*e.g.*, in the case of reference types). As an example of this distinction, consider these two functions:

```
fn foo<U>(x: U) { }
fn call_foo<W>(x: W, y: W) { foo(&x); drop(x); foo(&y); }
```

`foo` is generic over a type `U`, taking an argument `x` of type `U`. `call_foo` calls `foo` on two references it creates. Semantically, the two borrows `&x` and `&y` that `call_foo` creates have different lifetimes `'a` and `'b`—in fact, their scope is disjoint, as `x` is `drop`ped before the lifetime of `y`'s borrow starts. Thus, in a RustBelt-style semantic interpretation of lifetimes, `'a` and `'b` do not even statically exist, as they are allocated during type checking, and consequently the same holds true for the semantic types `&'a W` and `&'b W` that the type parameter `U` of `foo` is instantiated with. On the other hand, the syntactic types that `foo` is monomorphized with *are* statically known: lifetimes are purely *ghost* and do not impact Rust's monomorphization.

The caller-callee interface by example. Dynamic parameters pose a key challenge for RefinedRust's caller-callee interface. Let us consider how it verifies the implementation of `call_foo` against trivial pre- and postconditions (as we have annotated no specification above). For that, let us first consider the specification of `foo`:[9] $\texttt{foo\_spec} := \lambda \mathit{USt}.\ \boldsymbol{\lambda}\ \mathsf{U}.\ \forall x.\ \{x @ \mathsf{U} \mid \top\}\{\top\}$.

The specification is parametric in the syntactic type *USt* for the type parameter `U`, as well as the semantic type parameter U (compatible with *USt*). The rest of the term gives the core functional specification: for any mathematical value x of the argument `x`, if the argument has the mathematical model x at semantic type U and the trivial precondition $\top$ is satisfied, then `foo` safely executes and ensures the trivial postcondition $\top$. When RefinedRust verifies this function, it proves:

$$\forall \mathit{USt}.\ \texttt{fn\_typed}\ (\texttt{foo\_impl}\ \mathit{USt})\ (\boldsymbol{\lambda}\ \mathsf{U}.\ \texttt{foo\_spec}\ \mathit{USt}\ \mathsf{U})$$

Here, `fn_typed` I S is RefinedRust's function typing judgement, saying that the source code I has the specification S. S may be polymorphic in semantic lifetime/type parameters (*e.g.*, over the type parameter U); internally `fn_typed` universally quantifies over them, such that an instantiation can be picked at any call to the function. In the concrete case, for any choice of syntactic type *USt*, we verify that `foo_impl` *USt* is well-typed for all semantic types compatible with *USt*.

Now concerning `call_foo`, since RefinedRust verifies it polymorphically, it does so for any compatible syntactic and semantic types for `W`. Since `call_foo` calls `foo`, RefinedRust furthermore assumes that `foo` has been monomorphized for the instantiation of its type parameter `U` with `&W` (*i.e.*, $U_{st} = \mathsf{RefSt}$), and that it satisfies the specification we verified above.[10] Formally, RefinedRust proves for `call_foo`:

$$\forall \mathit{WSt}, l_{foo}.\ l_{foo} \lhd_v (\boldsymbol{\lambda}\ \mathsf{U}.\ \texttt{foo\_spec}\ \mathsf{RefSt}\ \mathsf{U}) \mathrel{-\!\!*}$$
$$\texttt{fn\_typed}\ (\texttt{call\_foo\_impl}\ \mathit{WSt}\ l_{foo})(\boldsymbol{\lambda}\ \mathsf{W}.\ \texttt{call\_foo\_spec}\ \mathit{WSt}\ \mathsf{W})$$

Here, `call_foo_spec` is the specification of `call_foo` (trivial, similarly to `foo_spec`), l_{foo} is a memory location storing the monomorphized code of `foo`, and `call_foo_impl` is the source code of `call_foo`. Note that we need to assume the spec of `foo` to be generic in the semantic type U because it will be instantiated twice with two distinct semantic types, corresponding to `&'a W` and `&'b W`.

Updating the caller-callee interface to account for traits. Now, let us add a trait requirement `U: Iter` to `foo`. For simplicity, we will omit the handling of `Iter`'s attributes. For illustrative purposes, we also declare an implementation of the `Iter` trait for shared references; the implementation of `next` does not matter for the following discussion. This implementation is generic in the lifetime of the reference as well as the type of its referee. Finally, we update `foo` to require `U` to implement `Iter` and to call `next` on its argument.

[9] We omit the parameterization of generic functions by the mathematical type in the following, as it is not relevant for the discussion.

[10] Specifications for other functions are assumed (instead of directly dispatched with their proof) in order to allow for mutual recursion.

```
impl<'a, T> Iter for &'a T { type Item = (); fn next(&mut self) → Option<()> { ... } }
fn foo<U: Iter>(x: U) { x.next(); }
fn call_foo<W>(x: W, y: W) { foo(&x); drop(x); foo(&y); }
```

For the `&'a T` instance of `Iter`, RefinedRust generates instances `Iter_impl_ref_spec` (of type `Iter_spec`) and `Iter_impl_ref_attrs` (of type `Iter_attrs`), and proves that the former implies the base spec `Iter_base_spec` (specialized to `&'a T`) using `Iter`'s spec inclusion relation. (Note that, in this case, `Iter_impl_ref_spec` and `Iter_base_spec` do not significantly differ, as we did not provide a more specific specification to the impl. But this need not be the case in general.)

Function contract: Callee side. On the callee side, the specification `foo_spec` stays mostly the same, but also quantifies over the associated type Item now:

$$\texttt{foo\_spec} := \lambda \mathit{USt}, \mathit{ItemSt}.\ \boldsymbol{\lambda}\ \mathsf{U}, \mathsf{Item}.\ \forall x.\ \{x @ \mathsf{U} \mid \top\}\{\top\}$$

However, when verifying `foo` now, we also need to link its implementation against an implementation of `Iter::next` for U. One first attempt at stating that might be as follows, using the base specification `Iter_next_base_spec`:

(wrong attempt)

$$\begin{aligned}&\forall \mathit{USt}, \mathit{ItemSt}.\ l_{\mathit{iter\_next}} \lhd_v (\boldsymbol{\lambda}\ \mathsf{U}, \mathsf{Item}.\texttt{Iter\_next\_base\_spec}\ \mathit{USt}\ \mathit{ItemSt}\ \mathsf{U}\ \mathsf{Item}) \mathrel{-\!\!*}\\ &\texttt{fn\_typed}\ (\texttt{foo\_impl}\ \mathit{USt}\ \mathit{ItemSt}\ l_{\mathit{iter\_next}})\ (\boldsymbol{\lambda}\ \mathsf{U}, \mathsf{Item}.\ \texttt{foo\_spec}\ \mathit{USt}\ \mathit{ItemSt}\ \mathsf{U}\ \mathsf{Item})\end{aligned}$$

This specification leaves the requirement on `Iter::next` generic in the semantic types U and Item. However, this is too strong of an assumption: the implementation of `Foo` that the function is linked with may not be proven correct for *any* semantic type U—a case in point is the implementation of `Foo` for references, which only works for shared reference types! We have to *restrict* the implementation of `Iter::next` to only work for the semantic type that the function `foo` is actually instantiated with (*i.e.*, link up the two quantifiers over U). Recall that the obvious idea of pulling out the quantifier over U to the toplevel does not work, as semantic types are *dynamic*.

Our solution is to add an indirection by parameterizing over the specification of assumed traits and adding a *trait linking assumption* to the caller-callee interface. More concretely, we parameterize the proof of `foo` by the specification record `Iter_U_spec` of the instance of `Iter` that `foo` is instantiated with (parametric over an unknown generic signature `Iter_U_Args`, consisting of a number of lifetime and type parameters) and use the specification for `Iter::next` provided by this assumption. Then, in the verification we assume that there exists an instantiation I for the generic signature `Iter_U_Args` such that `Iter_U_spec` I implies the expected specification for `Iter` that `foo` assumes. We express the trait linking assumption via the notion of trait inclusion $\subseteq_{\texttt{Iter}}$ for `Iter`, and add the assumption to the specification of `foo` *below* the semantic quantifiers. A caller of `foo` has to prove the trait linking assumption at the call site, but it can do so in a context where the concrete instantiation of all of `foo`'s semantic parameters is known, and is thus able to pick a suitable instantiation for I.

On a more technical level, we express the addition of the trait linking assumption using the $S\,|_{\text{assume}}\,P$ operator, which adds P as a precondition to the specification S:

$$\begin{aligned}&\forall \mathit{USt}, \mathit{ItemSt}, \texttt{Iter\_U\_Args}, (\texttt{Iter\_U\_spec} : \texttt{generic\_spec Iter\_U\_Args Iter\_spec}).\\ &l_{\mathit{iter\_next}} \lhd_v (\boldsymbol{\lambda}\ I.\ (\texttt{Iter\_U\_spec}\ I).\texttt{Iter\_next\_spec}) \mathrel{-\!\!*}\\ &\texttt{fn\_typed}\ (\texttt{foo\_impl}\ \mathit{USt}\ \mathit{ItemSt}\ l_{\mathit{iter\_next}})\ (\boldsymbol{\lambda}\ \mathsf{U}, \mathsf{Item}.\\ &\quad (\texttt{foo\_spec}\ \mathit{USt}\ \mathit{ItemSt}\ \mathsf{U}\ \mathsf{Item})|_{\text{assume}}\ \exists\, I : \texttt{Iter\_U\_Args}.\ \texttt{Iter\_U\_spec}\ I \subseteq_{\texttt{Iter}}\\ &\quad\quad (\texttt{Iter\_base\_spec}\ \mathit{USt}\ \mathit{ItemSt}\ \mathsf{U}\ \mathsf{Item}))\end{aligned}$$

This indirection fixes the issues above, but fundamentally changes the function call contract. Therefore, let us look at the caller side, `call_foo`, again.

Function contract: Caller side. On the caller side, we now have to verify `call_foo` with an assumption on `foo` that can later be satisfied by a monomorphization of `call_foo` linked against the `Iter` impl for references. The key for that is to *partially instantiate* the assumption on `foo` to shared references, quantifying over the lifetime `'a` and semantic type W:

$$\forall WSt, l_{foo}.\ l_{foo} \lhd_v (\boldsymbol{\lambda}\ \texttt{'a}\ \mathsf{W},\ (\texttt{foo\_spec}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ (\&^{\texttt{'a}}\mathsf{W})\ ())) \mathrel{-\!\!*}$$
$$\texttt{fn\_typed}\ (\texttt{call\_foo\_impl}\ WSt\ l_{foo})(\boldsymbol{\lambda}\ \mathsf{W},\ \texttt{call\_foo\_spec}\ WSt\ \mathsf{W})$$

Function contract: Linking. Finally, let us observe how a monomorphized full program can be linked together: this is important, as we have to dispatch the trait linking assumptions here. As an example, assume that the `main` function is defined as follows:

```
fn main() { call_foo(42, 1337); }
```

In this case, we end up with the monomorphized functions `Iter::<'a i32>::next`, `foo<&'a i32>`, `call_foo<i32>`, and `main`. We allocate the corresponding monomorphized program code at four memory locations l_{Iter_next}, l_{foo}, l_{call_foo}, and l_{main}, having to show:

$$\texttt{fn\_typed}\ (\texttt{main\_impl}\ l_{call\_foo})\ \texttt{main\_spec} \tag{G_1}$$
$$* \texttt{fn\_typed}\ (\texttt{call\_foo\_impl}\ \mathsf{IntSt}_{i32}\ l_{main})(\texttt{call\_foo\_spec}\ \mathsf{IntSt}_{i32}\ \texttt{i32}) \tag{G_2}$$
$$* \texttt{fn\_typed}\ (\texttt{foo\_impl}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ l_{Iter\_next})\ (\boldsymbol{\lambda}\ \texttt{'a}, \mathsf{W}.\ \texttt{foo\_spec}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ (\&^{\texttt{'a}}\ \mathsf{W})\ ()) \tag{G_3}$$
$$* \texttt{fn\_typed}\ (\texttt{iter\_ref\_next\_impl}\ \mathsf{IntSt}_{i32})\ (\boldsymbol{\lambda}\ \texttt{'a}, \mathsf{W}. \tag{G_4}$$
$$(\texttt{Iter\_impl\_ref\_spec}\ \mathsf{IntSt}_{i32}\ \texttt{'a}\ \mathsf{W}).\texttt{Iter\_next\_spec})$$

Note that we instantiate the semantic parameters of the specs just enough to fulfill the requirements of the functions calling them, while not instantiating them fully: for instance, if we instantiated U in the specification of `foo` to $\&^{\texttt{'a}}$ i32 (because `foo` is only called on references of type `&i32`) instead of $\&^{\texttt{'a}}\mathsf{W}$ for a fresh W (as shown in (G_3), we could not fulfill the assumptions of the caller-side `call_foo` proof above (which still expects the spec to work for arbitrary semantic types W).

In order to show this goal, we employ Löb induction and use RefinedRust's rules for typing of function pointers (which encapsulates the guard posed by Löb induction), allowing us to assume:

$$l_{main} \lhd_v \texttt{main\_spec} * l_{call\_foo} \lhd_v (\texttt{call\_foo\_spec}\ \mathsf{IntSt}_{i32}\ \texttt{i32})$$
$$* l_{foo} \lhd_v (\boldsymbol{\lambda}\ \texttt{'a}, \mathsf{W}.\ \texttt{foo\_spec}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ (\&^{\texttt{'a}}\ \mathsf{W})\ ())$$
$$* l_{Iter\_next} \lhd_v (\boldsymbol{\lambda}\ \texttt{'a}\ \mathsf{W}.\ (\texttt{Iter\_impl\_ref\_spec}\ \mathsf{IntSt}_{i32}\ \texttt{'a}\ \mathsf{W}).\texttt{Iter\_next\_spec})$$

Now, the different conjuncts in the goal follow by applying the respective typing lemmas for the function and dispatching the requirements on other functions using the Löb induction hypothesis.

The interesting part is the proof for `foo` (G_3): we apply the callee-side lemma, instantiating `Iter_U_spec` to `Iter_impl_ref_spec`, U to $(\&^{\texttt{'a}}\ \mathsf{W})$, and Item to () in the lemma. Then, we first have to show that that the conclusion of the lemma implies G_3. Here, crucially, we have to dispatch the trait linking assumption, showing that the caller and callee agree on the `Iter` specification:

$$\texttt{fn\_typed}\ (\texttt{foo\_impl}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ l_{iter\_next})\ (\boldsymbol{\lambda}\ \texttt{'a}, \mathsf{W}.$$
$$(\texttt{foo\_spec}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ (\&^{\texttt{'a}}\ \mathsf{W})\ ())|_{\text{assume}}\ \exists\ \texttt{'c}, \mathsf{U}.\ \texttt{Iter\_impl\_ref\_spec}\ \texttt{'c}\ \mathsf{U} \subseteq_{\texttt{Iter}}$$
$$(\texttt{Iter\_base\_spec}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ (\&^{\texttt{'a}}\ \mathsf{W})\ ())) \mathrel{-\!\!*}$$
$$\texttt{fn\_typed}\ (\texttt{foo\_impl}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ l_{Iter\_next})\ (\boldsymbol{\lambda}\ \texttt{'a}, \mathsf{W}.\ \texttt{foo\_spec}\ \mathsf{RefSt}\ \mathsf{UnitSt}\ (\&^{\texttt{'a}}\ \mathsf{W})\ ())$$

This holds because the linking assumption can be satisfied by instantiating `'c` := `'a` and U := W, and using the fact that RefinedRust verifies every trait impl's spec to imply the correspondingly instantiated base spec.

Secondly, we have to show that `foo`'s assumption on `Iter::next` is satisfied for the instantiation above, *i.e.*, that the premise of the proof for `foo` is implied by the inductive hypothesis on `Iter::next`; this is trivial.

Summary. With this formulation of the caller-callee contract, the whole verification succeeds. In summary, the key changes to the contract include: (1) Adding *trait linking assumptions* to the caller-callee contract that ensure that the callee's trait assumptions can be met. (2) *Partially instantiating* the specification of callee functions a caller function links against, in order to instantiate the trait requirements. On a technical level, making these changes required developing a new representation of generic contexts for RefinedRust's type system, encoded using a variant of telescopes.

5.3 Soundness and Limitations

For the elaboration of traits (*i.e.*, the elaboration of user intent), we have to trust that Rust's trait resolution as implemented in `rustc` works correctly. However, we are always guaranteed to verify the same code that `rustc` compiles to machine code, as the trait resolution algorithm is the same.

For the encoding into Rocq, we have to trust the implementation of the RefinedRust frontend. However, encoding bugs are likely to generate type-checking errors in Rocq, which will subsequently be caught and prevent the verification from succeeding.

We have to further trust that the formalization of Radium in Rocq is a sufficiently accurate model of Rust's MIR semantics, as Radium forms the core source of trust for RefinedRust.

When verifying a closed program, the user need not trust our semantic model in Rocq, as one can always use RefinedRust's adequacy theorem for closed programs in order to obtain a proof of safety depending only on RefinedRust's operational semantics.

Our trait semantics currently does not support the following Rust features:

- GATs (Generic Associated Types), which allow associated types to themselves be generic.
- Trait requirements on associated types, which require associated types to themselves implement certain traits.

GATs would be relatively straightforward to support, but are fairly uncommon in Rust code. Trait requirements on associated types can potentially be cyclic and induce an infinite sequence of trait requirements, which would require techniques like step-indexing to resolve in our semantic model.

6 Evaluation

We evaluate RefinedRust with the goal of understanding its usability and proof-engineering cost when applied to realistic Rust code. In particular, we aim to answer the following questions: **Q1** How does the specification and manual proof effort of RefinedRust compare to existing non-foundational Rust verification tools that support iterators? **Q2** Does supporting higher-order Rust features such as traits, closures, and iterators increase the manual proof effort in practice?

6.1 Case Studies Ported from Creusot

To answer **Q1**, we ported Creusot's case studies on iterators [7] to RefinedRust and compared specification and proof sizes as well as verification time.

These case studies, listed in Fig. 7, are written purely in safe Rust and were originally used to evaluate Creusot's support for iterators across a range of algorithms that use iterators in various ways. To port them to RefinedRust, we removed Creusot-specific proof annotations (*e.g.*, proof asserts and ghost code) from their implementation, and ported the specifications from Creusot's specification language to RefinedRust's specification language.

In a number of places where vectors are being allocated, we had to add stronger preconditions: Rust's `Vec` implementation panics if the allocated size exceeds `isize::MAX` bytes, but both Creusot

Mod	LOC	Spec (RR)	Proof (RR)	Time (RR)	Spec (CR)	Proof (CR)	Time (CR)
`all_zero`	5	3	3	33.6s	3	0	0.41s/0.41s
`decuple_range`	7	3	2	14.21s	3	0	0.4s/0.39s
`skip_take`	4	2	0	29.97s	1	0	0.35s/0.35s
`concat_vec`	4	1	0	12.09s	1	0	0.39s/0.39s
`counter`	13	4	35	48.55s	4	0	6.86s/0.9s
`Hillel`	43	49	51	322.20s	93	16	7.75s/1.56s
`Knight's Tour`	85	56	59	635.01s	55	2	7.48s/1.5s

Fig. 7. Comparison of RefinedRust (RR) with Creusot (CR) on Creusot case studies. Lines of code (LOC) measured with `tokei` (for Rust/Creusot code) and `coqwc` (for Rocq code). Creusot proof times show initial proof time (cold start) and proof replay time (reusing a file describing the proof tree, common when working on a proof). Measurements made on a 2021 Macbook Pro with 64GB RAM and 10 cores.

and RefinedRust rule out panics. However, Creusot assumes specifications for vector operations that disregard this condition for convenience.[11] Conversely, we were able to strengthen the specifications for the Knight's Tour case study: Creusot's specification assumes that the chessboard the algorithm operates on is limited to size 10000 as this is more convenient for the SMT solver, which we replaced with the more precise condition that the square of the size is representable in the integer type.

Finally, since RefinedRust currently has limited support for Rust's slices, we had to add extra wrappers for encapsulating `Vec`'s iterator and indexing functionality.

For measuring the specification complexity, we have counted the lines of specification, including specifications for contained closures and loop invariants. Moreover, specifications include manually-written Rocq definitions used in the specification (in the case of RefinedRust) and logic-only definitions (in the case of Creusot). For measuring the proof complexity, we have counted the lines of manually-written Rocq proofs (in the case of RefinedRust) and lines of proof hints like proof asserts and ghost code embedded in the Rust code (in the case of Creusot).

We find that RefinedRust is competitive when it comes to specification complexity, benefiting from Rocq's vast standard library of definitions and expressive language. In terms of proof complexity, RefinedRust generally requires more manual proofs, while remaining within a reasonable margin of Creusot; here, Creusot benefits from the strong automation provided by its Why3 backend. As one might expect, RefinedRust is much slower on all of the examples, taking time in the order of minutes instead of seconds. This is an inherent disadvantage of RefinedRust coming from the different verification approaches: while Creusot trusts Rust's ownership type system and solves functional correctness conditions using a fast SMT solver, RefinedRust verifies both memory safety and functional correctness in Rocq.

6.2 Proof Effort in a Real-World Case Study

To answer **Q2**, we applied RefinedRust to verify core parts of a real-world system—the ACE security monitor [22]—that uses higher-order Rust features supported by RefinedRust. Our goal was to assess whether the abstractions introduced by RefinedRust are usable in practice and whether they lead to an increase in manual proof effort.

We focused on verification of core components of ACE's memory management subsystem and compared the manual verification effort required for code fragments that use different combinations

[11] Typically, the user is expected to run out of memory first before this triggers an unsoundness.

Function	LOC	Spec	Proof	Closures	Iterators	Higher-order	Unsafe
Page accessors	15	7	3	○	○	○	○
`Page::divide`	16	14	191	●	●	●	●
`Page::write`	9	10	25	○	○	○	●
`PA::largest_page_size`	15	17	36	○	○	○	○
`PA::add_memory_region`	25	27	55	●	○	◐	●
`PA::store_page_token`	24	13	239	○	○	○	○
`PA::acquire_page_token`	32	16	252	●	●	●	○
`PA::divide_page_token`	13	16	144	○	●	○	○
`PA::child_addr_and_size`	6	5	14	○	○	○	○

Fig. 8. All except for page accessors are using traits. `PA` is short for `PageAllocator`. LOC measured with `tokei` and `rocq wc` (excluding space and comments).

of Rust features. In particular, we verified ACE's page token abstraction (which includes the `divide_page` example from Fig. 1) and the page allocator.

The page allocator is by far the most challenging piece of real-world Rust code we have verified using RefinedRust. It is implemented as a recursive tree-based data structure, where each node corresponds to a well-aligned physical page and tree levels represent different page sizes. Correctness of the page allocator relies on deep functional correctness invariants that relate the logical allocation state of nodes and their subnodes. The definition of the main page allocator invariant alone (excluding auxiliary definitions) requires about 50 lines of Rocq code. Thus, interacting with these invariants requires non-negligible manual reasoning.

Feature usage. We evaluate the page allocator verification in this analysis because its implementation exercises all of RefinedRust's features. First, it contains several small `unsafe` code segments, for example, when page tokens are initially created from raw memory regions. Second, the implementation makes extensive use of Rust's higher-order iterator APIs, for instance for manipulating the vector of a node's children. We also include the verification of ACE's page token abstraction used by the page allocator. It handles much of the complexity related to reasoning about handling the memory ownership of a page and also uses `unsafe` code when, for example, adjacent page tokens are merged by the page allocator to reduce fragmentation.

Fig. 8 shows a selection of ACE functions we have verified, showcasing different combinations of features, as well as the resulting proof effort. We find that, broadly, the proof complexity is not necessarily related to the wealth of features used by a function, but is rather more affected by the underlying complexity of the reasoning. For instance, `store_page_token` and `acquire_page_token` are inverse operations requiring similar proof effort, but the former does not make use of iterators or closures, while the latter makes use of two different iterator chains. This indicates that using Rust's higher-order iterator features in RefinedRust does not cause an unreasonable overhead.

Proof effort. In total, we have verified around 600 lines of Rust code in ACE distributed across different modules, as shown in Fig. 9. Our verification required a small number of changes to the code in order for the verification to succeed: while we refactored the logical flow in several places in the code to be clearer as part of the verification process, we had to change only few lines of code specifically for the purpose of verification, and the Rust source code contains no artifacts of the proof except for the specifications itself. We believe it would also have been possible to verify the code without any changes, but this would have made the reasoning in some places

ACE SM component	LOC	Spec	Proof	Ghost state	Recursive types
Infrastructure (pointers utility, errors)	129	8	17	○	○
`ConfidentialMemoryAddress`	36	28	0	●	○
`NonConfidentialMemoryAddress`	34	25	0	●	○
`PageSize`	81	7	3	●	○
`Page`	103	66	369	●	○
`PageAllocator`	217	183	1066	●	●
Total	600	317	1455		

Fig. 9. On top of that in Rocq: 489 LOC of theory for Page size, 364 LOC for Page, 491 for Page allocator. LOC measured with `tokei` and `rocq wc` (excluding space and comments).

significantly more complicated. Our verification resulted in around 320 lines of specification, 1350 lines of supporting definitions, and 1460 lines of proofs.

Findings. In the process, we found two bugs in the page allocator. The first bug was related to incorrectly checking bounds in pointer arithmetic when initially creating page tokens in the page allocator. Although this bug was not exploitable in the current ACE implementation, future changes in other parts of the security monitor could have elevated it to a security vulnerability. The second bug caused part of the memory to be unintentionally discarded during initial creation of page tokens. Both bugs were not caught by manual inspection when writing formal specifications in the early stages of verification. We disclosed both bugs to the authors of ACE.

6.3 Discussion

In general, compared to non-foundational verification tools, RefinedRust trades off performance against foundational proofs and flexibility. While proof development with RefinedRust is thus inherently less responsive than with other tools (which give answers in seconds), we still think that verification of realistic code with RefinedRust is feasible. Performance engineering is a continuous problem for any realistic verification tool, and we think that there is likewise additional potential for performance optimizations in RefinedRust. Since the original version of RefinedRust [10], performance on its original case studies has improved by 70%, despite many generalizations.

Currently, the main bottlenecks in RefinedRust performance are: (1) Rocq Qed checking performance (on larger functions, Qed checking makes up about 40% of the total proof time), (2) Rocq unification/typechecking when applying lemmas in symbolic execution, and (3) sidecondition solver performance (`lia`/`nia`/custom solvers).

7 Related Work

Features in other Rust verifiers. Recursive types, traits, closures, and iterators have all been supported before by various Rust verification tools, with differing design choices and feature sets. However, none of these tools are foundational or directly support unsafe Rust code manipulating raw pointers.

First of all, Wolff et al. [28] show how to add support for closures to the Prusti verification tool [2]. In their implementation, closures have to be specified using first-order logic, and thus it is not possible to specify closures like the one used in `Page::divide` in §3, which can only be specified in separation logic. In follow-on work by Bílá et al. [5], the authors show how to use this support to specify and verify iterators in Prusti, including higher-order iterators like `map`. Our encoding of iterators is conceptually similar to Prusti's encoding, using a `Next` relation, but Prusti's encoding is

geared towards being highly automatable for SMT solvers. Our specification is more general, as it does not require specifying a `Done` relation (thus supporting iterators which are not fused), at the cost of being less easy to automate.

Similarly, Denis and Jourdan [7] show how to add support for traits, closures, and iterators to the Creusot verification tool [8]. Creusot uses elaborate encoding tricks to help the SMT solver infer invariants automatically and thus automate iterators to a very high degree, at the cost of complicated internal specifications (*e.g.*, the full specification of `Map` is over 100 lines long). Our style of specifying higher-order iterators is inspired by Creusot's specifications, but adds the ability to specify additional separation logic ownership to account for ownership not tracked by the Rust type system as shown in §3, at the cost of having to specify an invariant over the iterator. In some cases, we then have to manually derive an inductive property for it.

VeriFast's Rust frontend [9] is built to verify *safe abstraction* of Rust functions involving unsafe code, *i.e.*, to show that Rust functions adhere to their safety contract. VeriFast can show that individual functions of a trait implementation adhere to the safety contract, but has not been used for verification of functional correctness properties as expressive as in Creusot or RefinedRust.

GillianRust [3] is a recent Rust verification tool seamlessly supporting unsafe Rust in a similar way as RefinedRust. While it does not have support for iterators and closures right now, it is meant to be used in a hybrid approach together with Creusot (which supports iterators very well). As such, if the occurrences of unsafe code can be cleanly split from occurrences of iterators and closures, Creusot and GillianRust make for a compelling combination. However, as shown by our `divide_page` example, such a clean split is not always possible.

Verus [18] has made significant strides towards verifying systems code [17, 29]. Verus's approach to verification of unsafe code (now also adopted by Creusot [11]) is quite different from the approach taken by RefinedRust, VeriFast, or GillianRust: Verus and Creusot encode *linear ghost resources* into a combination of custom (zero-sized) Rust types and functional verification conditions, enabling it to handle ownership reasoning that Rust cannot handle with built-in types (like accesses to raw pointers). Thus, all of the ownership reasoning is encoded into Rust's type system. This requires changing the code to use the custom abstractions and manually passing around ghost resources, and while the Rust compiler can erase zero-sized ghost types in many cases, it is not always obvious that the instrumented code fully corresponds to the original code. This makes Verus suitable for verifying newly written code, which can be written to directly use Verus's wrapper libraries, but not for verifying existing code without significant modifications. Moreover, Verus does not support iterators and `for` loops in general, having only support for plain iterators (*e.g.*, on ranges and vectors) but not for general iterator combinators.

Creusot, Prusti, and Verus specify traits by adding new "ghost" members to Rust code at the specification site. This differs from RefinedRust's specification approach, which merely attaches specifications via Rust attributes and does not require modifying the Rust code itself, at the cost of a less accessible specification language.

Iterators in separation logic. Separation logic specifications for iterators have been studied extensively before [23, 16, 13, 21]. Pottier [23] specifies and verifies methods for iterating over hashtables in OCaml, using both *folds* and *iterators*. The style of specifying fold functions generically using separation logic—by picking a separation logic invariant about the fold state that the closure needs to preserve—is similar to our specification of `map`. However, their specification of iterators does not allow clients of iterators to mutate elements (as supported by mutable iterators in Rust).

Acknowledgments

We thank the anonymous OOPSLA 2026 reviewers for their helpful feedback. We also thank the anonymous CPP 2026 reviewers who gave extensive constructive feedback that enabled us to improve this paper.

Data Availability Statement

Accompanying this paper is an artifact [12] containing our new version of RefinedRust as well as the verification of the case studies from §6 and the examples from §2 using RefinedRust.

References

[1] 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.

[2] 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. doi:10.1145/3360573

[3] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimovic, and Philippa Gardner. 2025. A Hybrid Approach to Semi-automated Rust Verification. *Proc. ACM Program. Lang.* 9, PLDI (2025), 970–992. doi:10.1145/3729289

[4] Andrew Bresticker, Andy Dellow, Atish Patra, Atul Khare, Beeman Strong, Christian Bolis, Dingji Li, Dong Du, Dylan Reid, Eckhard Delfs, Fabrice Marinet, Guerney Hunt, Jiewen Yao, Kailun Qin, Manuel Offenberg, Nicholas Wood, Nick Kossifidis, Osman Koyuncu, Qing Li, Rajnesh Kanwal, Ravi Sahita, Rob Bradford, Samuel Ortiz, Steven Bellock, Vedvyas Shanbhogue, Wojciech Ozga, and Yann Loisel. accessed on August 2025. Confidential VM Extension (CoVE) for Confidential Computing. https://github.com/riscv-non-isa/riscv-ap-tee/releases/download/v0.7/riscv-cove.pdf.

[5] Aurea Bílá, Jonas Hansen, Peter Müller, and Alexander J. Summers. 2022. Compositional Reasoning for Side-effectful Iterators and Iterator Adapters. doi:10.48550/ARXIV.2210.09857

[6] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. doi:10.1145/3371102

[7] Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators. In *TACAS (2) (LNCS, Vol. 13994)*. Springer, 93–110. doi:10.1007/978-3-031-30820-8_9

[8] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *ICFEM (Lecture Notes in Computer Science, Vol. 13478)*. Springer, 90–105. doi:10.1007/978-3-031-17244-1_6

[9] Nima Rahimi Foroushaani and Bart Jacobs. 2022. Modular Formal Verification of Rust Programs with Unsafe Blocks. arXiv:2212.12976 [cs.LO] https://arxiv.org/abs/2212.12976

[10] 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 (2024), 1115–1139. doi:10.1145/3656422

[11] Arnaud Golfouse, Armaël Guéneau, and Jacques-Henri Jourdan. 2026. Using Ghost Ownership to Verify Union-Find and Persistent Arrays in Rust. In *CPP*. ACM, 383–397. doi:10.1145/3779031.3779086

[12] Lennard Gäher, Sascha Kehrli, Vincent Lafeychine, Avraham Shinnar, Wojciech Ozga, Guerney Hunt, and Derek Dreyer. 2026. Artifact for Bringing Foundational Verification to Real-World Rust Code. doi:10.5281/zenodo.21636474

[13] Christian Haack and Clément Hurlin. 2009. Resource Usage Protocols for Iterators. *J. Object Technol.* 8, 4 (2009), 55–83. doi:10.5381/JOT.2009.8.4.A3

[14] Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-order modules and the phase distinction. In *POPL*. Association for Computing Machinery, New York, NY, USA, 341–354. doi:10.1145/96709.96744

[15] 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. doi:10.1145/3158154

[16] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. 2009. Design patterns in separation logic. In *TLDI*. ACM, 105–116. doi:10.1145/1481861.1481874

[17] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *SOSP*. ACM, 438–454. doi:10.1145/3694715.3695952

[18] 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. doi:10.1145/3586037

[19] 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. doi:10.1145/3519939.3523704

[20] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 15:1–15:54. doi:10.1145/3462205

[21] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: dependent types for imperative programs. In *ICFP*. ACM, 229–240. doi:10.1145/1411204.1411237

[22] Wojciech Ozga, Guerney D. H. Hunt, Michael V. Le, Gaeher Lennard, Avraham Shinnar, Elaine R. Palmer, Hani Jamjoom, and Silvio Dragone. 2025. ACE: Confidential Computing for Embedded RISC-V Systems. https://arxiv.org/pdf/2505.12995.

[23] François Pottier. 2017. Verifying a hash table and its iterators in higher-order separation logic. In *CPP*. ACM, 3–16. doi:10.1145/3018610.3018624

[24] 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. doi:10.1145/3453483.3454036

[25] 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.

[26] The Rust Team. 2020. The Rust programming language. https://rust-lang.org.

[27] Linus Torvalds. 2022. Merge Commit for initial Rust support in the Linux kernel. Last accessed 24 Oct 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b

[28] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. doi:10.1145/3485522

[29] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. 2024. VeriSMo: A Verified Security Module for Confidential VMs. In *OSDI*. USENIX Association, 599–614. https://www.usenix.org/conference/osdi24/presentation/zhou

Received 2026-03-17; accepted 2026-06-10