Verus: Verifying Rust Programs using Linear Ghost Types

ANDREA LATTUADA*, VMware Research, Switzerland
TRAVIS HANCE, Carnegie Mellon University, USA
CHANHEE CHO, Carnegie Mellon University, USA
MATTHIAS BRUN, ETH Zurich, Switzerland
ISITHA SUBASINGHE†, UNSW Sydney, Australia
YI ZHOU, Carnegie Mellon University, USA
JON HOWELL, VMware Research, USA
BRYAN PARNO, Carnegie Mellon University, USA
CHRIS HAWBLITZEL, Microsoft Research, USA

The Rust programming language provides a powerful type system that checks linearity and borrowing, allowing code to safely manipulate memory without garbage collection and making Rust ideal for developing low-level, high-assurance systems. For such systems, formal verification can be useful to prove functional correctness properties beyond type safety. This paper presents Verus, an SMT-based tool for formally verifying Rust programs. With Verus, programmers express proofs and specifications using the Rust language, allowing proofs to take advantage of Rust's linear types and borrow checking. We show how this allows proofs to manipulate linearly typed permissions that let Rust code safely manipulate memory, pointers, and concurrent resources. Verus organizes proofs and specifications using a novel mode system that distinguishes specifications, which are not checked for linearity and borrowing, from executable code and proofs, which are checked for linearity and borrowing. We formalize Verus' linearity, borrowing, and modes in a small lambda calculus, for which we prove type safety and termination of specifications and proofs. We demonstrate Verus on a series of examples, including pointer-manipulating code (an xor-based doubly linked list), code with interior mutability, and concurrent code.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Rust, linear types, systems verification

ACM Reference Format:
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, Article 85 (April 2023), 30 pages. https://doi.org/10.1145/3586037

*Research work done mainly at ETH Zurich, Switzerland.
†Research work done as a student at the University of Melbourne, Australia, and as a research assistant at ETH Zurich.

Authors' addresses: Andrea Lattuada, VMware Research, Switzerland, lattuada@vmware.com; Travis Hance, Carnegie Mellon University, USA, thance@andrew.cmu.edu; Chanhee Cho, Carnegie Mellon University, USA, chanheec@andrew.cmu.edu; Matthias Brun, ETH Zurich, Switzerland, matthias.brun@inf.ethz.ch; Isitha Subasinghe, UNSW Sydney, Australia, i.subasinghe@unsw.edu.au; Yi Zhou, Carnegie Mellon University, USA, yizhou5@andrew.cmu.edu; Jon Howell, VMware Research, USA, howell@vmware.com; Bryan Parno, Carnegie Mellon University, USA, parno@cmu.edu; Chris Hawblitzel, Microsoft Research, USA, chris.hawblitzel@microsoft.com.

2475-1421/2023/4-ART85
https://doi.org/10.1145/3586037

1 INTRODUCTION

The Rust programming language [Klabnik and Nichols 2018; Matsakis and Klock 2014] has brought linear types into the mainstream. Rust's sophisticated type system incorporates linear types and borrowing, making it possible to write low-level systems code in a type-safe way without requiring garbage collection. This makes Rust an attractive language for developing low-level software that needs both high performance and high assurance, and Rust has gained rapid acceptance for system programming over the last few years [Google Security Blog 2021; Vaughan-Nichols 2022].

Nevertheless, even type-safe code can still contain bugs that harm a program's security and reliability. Furthermore, systems programmers using Rust sometimes resort to unsafe code (via Rust's `unsafe` keyword) for programming styles that do not fit into Rust's linearity discipline; e.g., it is awkward to encode doubly linked lists in Rust because the backwards links violate linearity.

Formal verification promises to prove deeper properties about Rust programs, including properties about low-level code that would otherwise require unsafe Rust features. Hence, we introduce Verus, an SMT-based tool for verifying Rust code. SMT-based verification can help Rust overcome the limitations of Rust's strict typing discipline, making it possible to safely express low-level code like doubly linked lists or safe implementations of reader-writer locks for concurrent code.

Just as importantly, we argue that Rust's linear type system can help make SMT-based verification easier, bringing the power of substructural logics, like concurrent separation logic [O'Hearn 2007; Reynolds 2002], to SMT-based reasoning. In particular, we demonstrate the use of linear ghost permissions that enable a program to take specific actions on specific resources, such as writing to a memory location. Since the permissions are linear, they can track the evolving state of a resource in the same way that separation logic formulas can track the state of a resource. Since the permissions are ghost, they exist only during type checking and verification, and do not impose any overhead on compiled executable code.

To take advantage of Rust's type system for checking linear ghost permissions, Verus uses Rust to express specifications and proofs, running Rust's linearity and borrow checking on the proofs. By using a single language for specifications, proofs, and executable code, Verus follows in the footsteps of earlier frameworks that combine proofs and programming, such as Coq [Coq Development Team 2022], Dafny [Leino 2010], F* [Swamy et al. 2016], and Lean [de Moura et al. 2015]. However, these systems were designed from scratch for verification, unlike Rust, which was designed strictly as a programming language. Therefore, it is important to ensure that the subset of Rust that Verus allows in proofs and specifications is sound as a proof language.

In particular, we need to make sure that proofs terminate and that functions used in specifications are pure, mathematical functions, whereas executable code might contain infinite loops, be nondeterministic, or have side effects. In a language like Rust that contains recursive types, higher-order functions, and type classes (traits), termination can be particularly subtle: without positivity restrictions on recursive type definitions, for example, these features together can encode nontermination. Nevertheless, we want to restrict recursive types only for specifications and proofs, not for executable code, unlike Coq, Dafny, and Lean, which restrict all recursive types.

To enforce the distinction between specifications, proofs, and executable code, Verus introduces a mode system that classifies all code as specification, proof, or executable, where specification and proof code are checked for termination. All three modes of code are type checked; proof and executable code are checked for linearity and borrowing; only executable code is compiled to machine code. We formalize this mode system using a small Rust-inspired lambda calculus, proving preservation and progress for all well-typed expressions and termination for all specification and proof expressions (Section 10).

Verus currently supports a large set of proof features and a large subset of ordinary Rust features:

- Rust's finite range integers (i32, u32, etc.) as well as infinite-range integers (int, nat) for specifications and proofs
- recursive algebraic datatypes (structs and enums), including mutual recursion, pattern matching, and pattern match guards
- mutable variables, while loops, and return statements
- recursive specification functions and inductive proofs, including mutual recursion and lexicographic decreases clauses
- passing function arguments by borrowing (both `&` and `&mut`)
- lifetime parameters on structs
- generics (parametric polymorphism), with support for simple traits (type classes)
- first-class functions (Rust closures) in specifications
- modules with public and private definitions
- preconditions, postconditions, and loop invariants
- quantifiers (forall, exists, choose) with both automated and manual SMT trigger selection, as well as integrated quantifier profiling to diagnose verification performance issues
- programmer control of SMT performance by selectively hiding and revealing specification function definitions, including control over recursive function unrolling
- support for bit-vector reasoning and proof by computation
- strings and characters
- a library of types for specifications, including sequences, sets, and maps
- low-level pointer reasoning
- concurrency and state machines

Verus is also able to handle some situations that require `unsafe`. For example, Verus is able to verify the use of raw pointers and unsafe cells, which can be useful for some low-level pointer reasoning, lock implementations, and interior mutability use-cases. As we will see, Verus' support for linear ghost state is crucial for this support. Note though, that Verus does not attempt a full aliasing and provenance model for Rust's pointers; our simplified model for "raw pointers" only handles those that point into the global heap.

Some features are still missing, notably support for separate verification of multiple crates and functions that return mutable references. However, we believe that supporting crates and various other Rust features is a matter of engineering, and supporting functions that return mutable references can follow earlier research (Section 11) by Prusti [Astrauskas et al. 2022], RustHorn [Matsushita et al. 2020], Creusot [Denis et al. 2022], and Aeneas [Ho and Protzenko 2022].

Regardless, this paper will not focus on all the features supported by Verus, but will instead focus on the most novel contributions:

(1) usage of Rust's linearity and borrow checking in proofs
(2) verification of pointer-manipulating Rust code and concurrent Rust code, based on a combination of linearity, borrowing, and SMT solving
(3) a mode system for enforcing the different properties of specs, proofs, and executable code
(4) formalization of the mode system, including checking of linearity and borrowing

The remainder of this paper introduces Verus by example (Section 2); discusses handling unsafe code (Section 3); applies Verus to pointer-based code (including a doubly linked list, Section 4), interior mutability (Section 5), and concurrent code (Section 6); discusses Verus' implementation (Section 7), user experience (Section 8), and limitations (Section 9); and presents syntax, semantics, and proofs for a formal lambda calculus with modes, linearity, and borrowing (Section 10).

```
#[spec] fn fibo(n: nat) -> nat {
  decreases(n); Ⓐ
  if n == 0 { 0 }
  else if n == 1 { 1 }
  else { fibo(n - 2) + fibo(n - 1) }
}

#[proof] fn lemma_fibo_is_monotonic(i:nat, j:nat) {
  requires(i <= j); Ⓑ spec
  ensures(fibo(i) <= fibo(j)); Ⓒ spec
  decreases(j - i); Ⓓ spec

  if (i < 2 && j < 2) || i == j {
  } else if i == j - 1 {
    reveal_with_fuel(fibo, 2); Ⓔ
    lemma_fibo_is_monotonic(i, j - 1); Ⓕ
  } else {
    lemma_fibo_is_monotonic(i, j - 1);
    lemma_fibo_is_monotonic(i, j - 2);
  }
}

#[spec] fn fibo_fits_u64(n: nat) -> bool {
  fibo(n) <= u64::MAX
}

#[exec] fn fibo_impl(n: u64) -> u64 {
  requires(fibo_fits_u64(n)); Ⓖ spec
  ensures(|result: u64| result == fibo(n)); Ⓗ spec

  if n == 0 { return 0; }
  let mut prev: u64 = 0;
  let mut cur: u64 = 1;
  let mut i: u64 = 1;
  while i < n {
    invariant([ Ⓘ spec
      0 < i && i <= n,
      fibo_fits_u64(n as nat),
      fibo_fits_u64(i as nat),
      cur == fibo(i),
      prev == fibo(i as nat - 1),
    ]);
    let new_cur = cur + prev;
    prev = cur;
    cur = new_cur;
    assert(prev == fibo(i as nat)); Ⓙ proof
    i = i + 1;
    lemma_fibo_is_monotonic(i, n); Ⓚ proof
  }
  cur
}
```

Fig. 1. A proof of correctness of a function computing the n-th Fibonacci number. We use circled letters, similar to Ⓩ, to mark points of interest in the code. The markers spec and proof indicate specification and proof mode code respectively when it differs from the mode of the function.

2 VERUS BY EXAMPLE

This section introduces the basic features of Verus by walking through a simple example that computes Fibonacci numbers, shown in Figure 1. The example consists of a set of functions written in Rust. Each function is annotated with an attribute, using Rust's `#[...]` attribute syntax, to indicate whether the function is executable code (`#[exec]`), proof code (`#[proof]`), or specification code (`#[spec]`). We refer to `exec`, `proof`, and `spec` as modes; Figure 2 summarizes the properties of these three modes.

In Verus, specifications and proofs are simply Rust code, parsed with Rust's parser and checked with Rust's type checker. This avoids the need for systems programmers to learn a separate verification language, making verification more accessible and convenient. It also allows specifications and proofs to take advantage of Rust's features, such as recursive functions, arithmetic, algebraic datatypes, pattern matching, modules, closures, traits, etc. For soundness's sake, Verus places some limits on the features that specifications and proofs can use. In particular, specifications must be deterministic, and recursive `spec` functions and recursive `proof` functions must terminate.

The Verus tool, which extends the Rust compiler, erases all ghost code (all specifications and proofs) before the code is compiled to machine code. In the example, `lemma_fibo_is_monotonic`, `fibo`, and `fibo_fits_u64` are all erased before compilation. Furthermore, the executable function `fibo_impl` contains small bits of specification and proof inside its body (Ⓖ, Ⓗ, Ⓘ, Ⓙ, Ⓚ), and this ghost code is also erased.

Verus encodes preconditions and postconditions as calls to Verus library functions named `requires` and `ensures`. Postconditions may refer to the return value; for this, the ensures function accepts a Rust closure that declares a name for the return value. (In Rust, first-class functions are called closures and have the syntax "`|...parameters...| body`".) The example uses a postcondition to prove that the executable function `fibo_impl` computes the same result as the mathematical definition of the n^{th} Fibonacci number in the `fibo` function: the `ensures` clause Ⓗ establishes this postcondition for `fibo_impl`. Because the return value is a bounded 64-bit unsigned integer, `fibo_impl` can only accept a parameter `n` such that the n-th Fibonacci number fits in the type of

the return value: this is established by the `requires` clause Ⓖ. Note that Verus extends Rust's type system with two new integer types, `int` (mathematical integers $\mathbb{Z}$), and `nat` (natural numbers $\mathbb{N}$), so that specifications and proofs can talk about arbitrary integers. Executable code, however, is limited to Rust's finite-width integer types like `u64`, since `int` and `nat` aren't compilable to machine code.

To help prove the postcondition, the `fibo_impl` function uses a loop invariant Ⓘ containing a list of clauses that must be true before and after each loop iteration. Given preconditions, postconditions, and loop invariants, Verus uses standard weakest precondition reasoning [Dijkstra 1975] to generate a verification condition for `fibo_impl`. It then sends this verification condition to the Z3 SMT solver [de Moura and Bjørner 2008].

In many cases, the SMT solver can prove the verification condition completely automatically. In other cases, the proof may require reasoning beyond the SMT solver's abilities. For example, to prove the absence of 64-bit integer overflow, `fibo_impl` relies on the Fibonacci sequence being monotonic, which requires an inductive proof that the SMT solver cannot generate automatically. Instead, the programmer supplies an inductive proof in the form of a recursive `proof` function (e.g., `lemma_fibo_is_monotonic`). The programmer can also add explicit assertions Ⓙ that serve as hints to the SMT solver. This style of SMT-based verification with programmer-supplied lemmas and hints is similar to other verification systems like Boogie [Barnett et al. 2005], Dafny, and F*.

To improve verification performance, Verus strives to keep the verification condition encoding lightweight, so that the SMT encoding of specifications is not much larger than the original specifications written in Verus code. In particular, calls to `spec` functions are translated directly into calls to SMT functions, with no additional overhead. For this reason, Verus `spec` functions are total functions that do not have preconditions and postconditions. This design is similar to Boogie, though it differs from Dafny and F*.

This design choice has a downside, since precondition failures can provide the developer with early feedback to find errors in specification functions and in how they are used. In order to restore that feedback, Verus introduces `recommends` clauses: soft preconditions for `spec` functions, which Verus only considers when there is a verification error. At that point, it performs a separate check for soft preconditions of `spec` functions that are mentioned in the context of the failure, and reports failures as warnings for the developer.

Recursive `spec` functions and recursive `proof` functions are valid only if they terminate on all inputs (otherwise, they could encode unsound circular reasoning). Verus requires that all such functions contain a `decreases` clause Ⓐ Ⓓ and each recursive call must decrease the expression in the clause. The recursive definition of the n^{th} Fibonacci number Ⓐ in Figure 1 is legal because both recursive calls decrease the expression `n`. (Verus also imposes positivity restrictions on recursive type definitions to prevent nontermination, as discussed in Section 10.) The SMT solver may need to unfold definitions of a recursive `spec` function. As in Dafny and F*, Verus uses an integer "fuel" to control the number of unfoldings. The `reveal_with_fuel` Ⓔ function controls the fuel level.

2.1 Linearity, Borrowing, Spec Variables, and Proof Variables

Rust types are linear by default: unless a type implements the Rust `Copy` trait, values of the type can only be moved from one variable to another, not copied. For example, the Rust `Vec<T>` type for vectors is linear. The following code is illegal in Rust because it attempts to duplicate a `Vec<u64>` value, returning both copies of the value in a pair:

```
#[exec] fn f(v: Vec<u64>) -> (Vec<u64>, Vec<u64>) {
  let v1 = v;
  let v2 = v; // illegal, tries to duplicate v
  (v1, v2)
}
```

	specification mode	proof mode	executable mode
compiled or ghost	ghost	ghost	compiled
code style	purely functional	mutation allowed	mutation allowed
linearity & borrowing checking	not checked	checked	checked
can call specification functions	yes	yes	yes
can call proof functions	no	yes	yes
can call executable functions	no	no	yes
determinism	deterministic	nondeterministic	nondeterministic
termination	must terminate	must terminate	nontermination ok
preconditions/postconditions	none	requires/ensures	requires/ensures

Fig. 2. Summary of Verus' modes and their properties.

On the other hand, Rust code can duplicate immutable references to values, as long as the scope of the references is limited. In Rust terminology, a reference of type `&T` temporarily *borrows* from an owned value of type `T`. During the borrowing, the original owned value of type `T` is inaccessible. When the references go out of scope, the original owned value becomes accessible again. Rust code can also borrow a mutable reference of type `&mut T`; in contrast to immutable references, mutable references cannot be duplicated. Rust enforces the property that a value cannot be borrowed both immutably and mutably simultaneously.

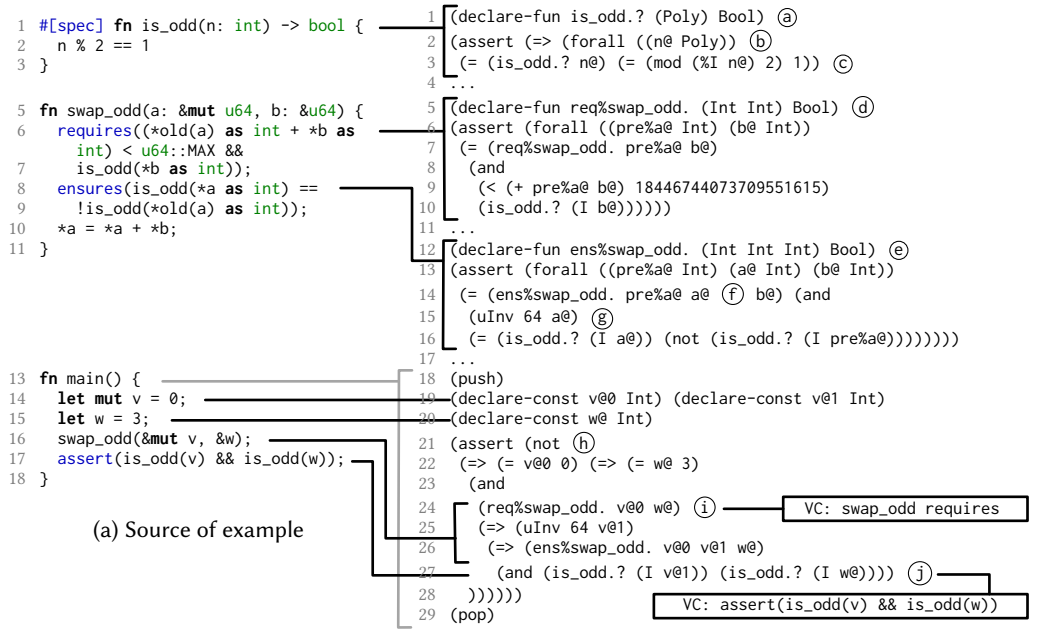
Rust contains a sophisticated "borrow checker" that checks linearity and borrowing. Similar to Creusot [Denis et al. 2022], Verus trusts the results of Rust's borrow checker, and does not attempt to recheck these results in the SMT solver, since this would just slow down the SMT solving. Because of this, Verus can rely on the properties of borrowing in its SMT encoding. For example, Verus encodes immutably borrowed references `&T` and owned heap pointers (`Box<T>`, `Rc<T>`, and `Arc<T>`) simply as values of type `T`, not as pointers to locations.

Verus specifications, in contrast to ordinary Rust code, are not checked for linearity and borrowing; specifications can freely copy any value of any type. This allows specifications to freely talk about linear values, potentially mentioning a single linear variable multiple times in a precondition or postcondition, for example. Verus code can also store nonlinear copies of linear variables inside *spec variables*, declared with the attribute `#[spec]`:

```
#[exec] fn f(v: Vec<u64>) {
  #[spec] let v1 = v; // copies v into spec variable v1
  #[spec] let v2 = v; // copies v into spec variable v2
  assert(v1.len() == v2.len());
}
```

Spec variables are similar to ghost variables in Dafny or erased values in F*. However, Verus also supports *proof variables*, declared as `#[proof]`, which do not have a correspondence in Dafny or F*. Proof variables sit midway between `exec` variables and `spec` variables: like `spec` variables, they are ghost and are not compiled to machine code, but like `exec` variables, they are checked for linearity. By default, variables in `exec` functions are `exec`, while variables in `proof` functions are `spec` unless declared `#[proof]`. Spec functions can only use `spec` variables, not `proof` variables or `exec` variables.

Since `proof` variables are both linear and ghost, they can represent abstract linear permissions to perform operations, which can produce and consume the linear permissions. The next section describes how Verus exploits this feature to verify safe low-level pointer manipulation that would, in ordinary Rust, require unsafe code. (In fact, Verus does not support verification of code marked with Rust's `unsafe` keyword; instead, its goal is to provide safe replacements for unsafe Rust features, based on linear ghost permissions and SMT-based verification.)



(a) Source of example

(b) SMT-LIB encoding of the verification conditions for `main`.

Fig. 3. A simple example program and relevant parts of its encoding in Z3. The SMTLIB encoding has been slightly simplified to aid readability, but without compromising accuracy. In particular, we rename the constants, and we elide the patterns chosen for quantifier instantiation in Z3 [de Moura and Bjørner 2007], some temporary variables used to optimize the SMT encoding, and some facilities for error reporting.

2.2 Simplifying Verification Conditions with Linear Types

Potential aliasing of variable bindings in the presence of mutation complicates verification [Borgida et al. 1995] because it requires explicitly reasoning about memory to determine the potential effects of each program statement. Given two bindings `p` and `q`, a predicate `P(p)` about the data reachable from `p`, and a statement `S[q]` which mutates one of the memory locations reachable from `q`, if `P(p)` is true before `S[q]`, then `P(p)` is guaranteed to remain true after `S[q]` if all the memory locations reachable from `p` and `q` are disjoint. Verus relies on the properties enforced by Rust's "borrow checker" to avoid explicit memory reasoning: when encoding `proof` and `exec` function bodies Verus treats the data associated with a uniquely-owned binding as an immutable value. Mutable bindings are represented with single static assignment to immutable SMT constants, one after each mutation. We discuss Verus' encoding strategy with an example.

Figure 3 shows a simple program, and how Verus encodes it into SMT-LIB [Barrett et al. 2010], the input to Z3. First let us dispatch some boilerplate that clutters the figure. The functions `%I`, `I` and the sort `Poly` appear often. They are part of the polymorphism encoding machinery of Verus, which is inspired by Boogie[Leino and Rümmer 2010]. Function `I` is a cast from `Int` to `Poly`, a Z3 sort representing a polymorphic type, and `%I` is a cast from `Poly` to `Int`. `spec` function arguments are always `Poly` due to interactions with the Z3's quantifier instantiation. `uInv 64` ⓖ is a typing invariant that restrict the SMT `Int` type to the range of Rust's `u64` machine type.

In SMT-LIB, functions are defined by constructing axioms (e.g. ⓑ) that relate their declaration (e.g. ⓐ) to their definition (e.g. ⓒ). Verus' `spec` is designed to closely match SMT logic, enabling the straightforward encoding of `is_odd` ⓒ. Similarly, the SMT functions representing the precondition and postcondition for `swap_odd` (`req%swap_odd.` ⓓ and `ens%swap_odd.` ⓔ respectively)

closely match their corresponding spec-mode Verus code. The mutable reference `a: &mut u64` is represented as a pair of constants, `pre%a@` and `a@` (f), respectively the initial and final value. The immutable reference (`b: &u64`) is represented as a single constant `b@`. Reference types (`&mut` and `&`) do not need special treatment thanks to the borrow checker's guarantees: for example the arguments `a` and `b` cannot be aliased because mutable and immutable references to the same data cannot exist at the same time.

The last SMT-LIB fragment is the encoding of the `main` function and its associated verification conditions. Like other tools, Verus encodes its proof search as a query to Z3 to find an assignment to constants that falsifies (h) verification conditions: an `unsat` result is a proof that such assignment does not exist, i.e. verification succeeded. The constants `v@0` and `v@1` represent the value of binding `v` before and after the call to `swap_odd`, and `w` represents the immutable binding `w`. The call to `swap_odd` is encoded modularly with a verification condition to check its precondition (i).

Another call to `ens%swap_odd.` introduces its postcondition as an antecedent for all future verification conditions in the function. Thanks to Rust's linear type system, there is no need to explicitly model the heap here: the two arguments to `swap_odd` are guaranteed to point to distinct regions of memory. Finally, the `assert` is encoded as part of the verification condition (j).

3 HANDLING UNSAFE CODE SAFELY

Unsafe code can often be a sticking point for users seeking strong guarantees about their Rust program, as it has the potential to undermine Rust's famous memory-safety guarantees. In particular, the Rust language guarantees that a program that does not use `unsafe` code must unconditionally be memory-safe; however, when `unsafe` code is used, the program becomes *conditionally memory-safe*; i.e., the program is only memory-safe if the program obeys certain rules when using `unsafe`.

Verus supports a few trusted primitives that are conditionally memory-safe in this manner. For these cases, their correctness conditions are encoded as Verus specifications. Therefore, users can be sure that their code is truly memory-safe (in addition to Verus' other guarantees) as long as Verus' SMT verification proves that the code upholds the contracts.

As a simple example, consider the operation of indexing into a vector: one of the most ubiquitous operations in all of software, yet also one of the most fraught for memory-safety violations. In Rust, indexing into a vector always performs a bounds check; it is memory-safe because it will always panic rather than access memory-out-of-bounds. On the other hand, Rust's `get_unchecked` does not perform any bounds check, and therefore it an `unsafe` function. That is, `get_unchecked` is conditionally memory-safe because it is only safe if the user calls the function with a valid index.

We can write this condition as a Verus specification, and provide `get_unchecked` as a trusted primitive function:

```
1 fn safe_get_unchecked<V>(v: &Vec<V>, i: usize) -> &V {
2   requires 0 <= i && i < v.len()
3   ...
4 }
```

In the next sections, we will see some more advanced examples.

4 SAFE POINTER MANIPULATION WITH LINEAR GHOST TYPES

4.1 Low-Level Pointer Manipulation with Linear Ghost Permissions

While Rust's reference types and borrowing rules provide a memory-safe framework for many use cases, they are sometimes insufficiently expressive and *raw pointers* may be required. For example, a doubly-linked list, where each node may be pointed to by two neighbors, violates the unique-ownership discipline of Rust. Raw pointers are one way to work around this, although dereferencing raw pointers in Rust requires `unsafe` code. Verus supports raw (heap) pointers. The

most notable aspect of this support is that, in order to provide a specification to enforce memory safety, we need to make use of linear ghost state.

Specifically, Verus introduces a core primitive `PPtr<T>` ("permissioned pointer") as a zero-cost alternative to raw heap pointers, along with an associated type `PermData<T>` ("permission plus data") which is to be used in `proof` mode, i.e., they are linear ghost objects as discussed in the previous section. Calls to the `PPtr<T>` API require ownership of this ghost permission object in order to dereference the pointer, which prevents data races and other forms of access that are undefined behavior in Rust's memory model.

However, the `PermData<T>` object does not "just" have the role of maintaining memory safety; it also tracks the data behind the pointer. Tracking permissions and data this way lets us write proofs in a style similar to that of separation logics. Specifically, the permissions object has two fields. The first, `perm.view().pptr`, indicates the pointer that the permission object corresponds to, and the second, `perm.view().value`, gives the data behind the pointer. The value field is an `Option<T>`, where a value of `Some(v)` means the memory stores `v`, and a value of `None` indicates that the memory is uninitialized. (This should not be confused for the *runtime representation* of an `exec`-mode `Option<T>`, where `None` is a legitimate, initialized value.)

Figure 4a shows two key functions from the `PPtr` API: a function to write through the pointer (`write`) and a function to read through it (`read`). Both functions require that the permission is actually associated with the pointer being dereferenced Ⓛ Ⓝ and `read` requires that the memory being read from is in an initialized state Ⓞ. Meanwhile, `write`'s postcondition Ⓜ says that the updated permission object contains the written value, while `read`'s postcondition Ⓟ says that the returned value is the value tracked via the permission. Figure 4b illustrates the usage of `write` and `read`, together with allocation and deallocation, showing how the permission `value` is updated.

It is crucial that the `proof`-mode object `PermData<T>` obeys Rust's ownership rules. For example, Figure 4 shows how this prevents a use-after-free bug. When we free the pointer's memory Ⓠ, the `perm` variable is consumed. Thus Rust's linearity checker would report an error if the code attempted to read the pointer again Ⓡ, as this produces another use of `perm`.

Finally, observe that the safety of this API depends crucially on our ability to add preconditions (and validate them via the prover). For example, we saw that the specification of `write` requires that the permission correspond to the pointer being written through Ⓛ, and without this requirement, it would be wildly unsound. Thus, a safe API like this is *not possible to implement in vanilla Rust*: in order to be unconditionally safe, the precondition would need to be a run-time check, which would mean the `PermData` object could not be ghost, and the abstraction could not be zero-cost.

4.2 Verified Example: Doubly-Linked List

Rust's ownership model typically forces data structures to be acyclic, unless they use unsafe code. Here, we illustrate how `PPtr` can be used to verify data structures that have cyclic pointer arrangements by verifying a double-ended queue implemented with a doubly-linked list. Specifically, we use a doubly-linked list to represent a sequence $v_0, v_1, \ldots, v_{n-1}$, and we implement the four operations $\{\texttt{push}, \texttt{pop}\} \times \{\texttt{front}, \texttt{back}\}$. The ith node in the list has both a `prev` and `next` pointer alongside a single element of the sequence, v_i. The top level datatype, `DList`, contains `head` and `tail` pointers, pointing to the first and last nodes, respectively. The full version (which can be found in our supplementary materials [Lattuada et al. 2023b]) contains an additional space-saving optimization, where each node does not store its two pointers separately, but rather, stores their bitwise XOR.

Figure 5a shows the physical pointer structure of the list. However, the diagram does not properly reflect a valid ownership structure because it shows each node with multiple incoming pointers. In the Verus implementation, we include an additional field in `DList`: the ghost `permissions`

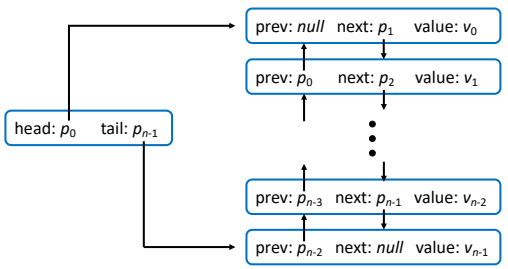
```
impl<T: Copy> PPtr<T> {
  // Equivalent of `*ptr = v`.
  #[exec] pub fn write(&self,
      #[proof] perm: &mut PermData<V>, v: V) {
    requires(equal(self.id(), old(perm).view().
    pptr)); (L)
    ensures([
      equal(perm.view().pptr, self.id()),
      equal(perm.view().value, Option::Some(v)),
      (M)
    ]);
    ...
  }

  // Read through the pointer and return the
    value. Requires the memory to be
    initialized.
  #[exec] pub fn read(&self,
      #[proof] perm: &PermData<V>) -> V {
    requires([
      equal(self.id(), perm.view().pptr), (N)
      perm.view().value.is_Some() ]); (O)
    ensures(|v: V| equal(Option::Some(v),
      perm.view().value)); (P)
    ...
  }
}
```

(a) Selected functions from the PPtr<T> API, a core Verus primitive.

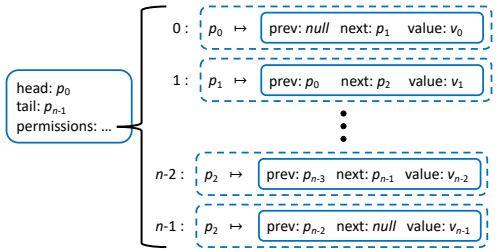
```
fn main() {
  // Allocate memory.
  let alloc = PPtr::<u64>::empty();
  // Unpack the return value into the pointer and
    the (ghost) permission
  let pptr = alloc.0;
  #[proof] let mut perm = alloc.1.0;

  // Initially, pptr points to unitialized memory,
    and the `perm` proof-object represents that as
    the value `None`.
  assert(equal(perm.view().pptr, pptr.id()));
  assert(equal(perm.view().value, Option::None));

  // We can write a value through the pptr (thus
    initializing the memory).
  pptr.write(&mut perm, 5);

  // Having written the value, this is reflected in
    the permission object:
  assert(equal(perm.view().value, Option::Some(5)));

  // We can now read it:
  let x = pptr.read(&perm);
  assert(x == 5);

  // Free the memory:
  pptr.free(perm); (Q)

  // This would error as `perm` was just consumed
  // let z = pptr.read(&perm); (R)
}
```

(b) Example usage of PPtr<T>

Fig. 4. The PPtr<T> API and an example usage. Though the two functions shown here require T: Copy, this is not a general restriction on the PPtr library.



(a) Physical pointer structure of a doubly-linked list.



(b) Ownership structure of a Verus doubly-linked list, which includes ghost state.

Fig. 5. Doubly-linked lists. The dashed boxes are ghost, proof-mode PermData objects.

field, which maintains permissions for *every* node in the doubly-linked list via a simple "flattened" structure, as in Figure 5b. Specifically, for each $i \in \{0, \ldots, n-1\}$, we maintain a PermData object that maps pointer p_i to the value it points to: the content of ith node, which contains v_i and the appropriate pointers, prev as p_{i-1} and next as p_{i+1}. To traverse the doubly-linked list, a user may use head to determine p_0, dereference p_0 using the 0th permission object, find p_1, and so on. In other words, we ghostily track the entire state of the list, but to get the same data in *exec-mode*, we need to actually walk the pointers.

Figure 6 shows a snippet of the API. The spec-mode view() function provides an abstraction of the list as a simple sequence $v_0, v_1, \ldots, v_{n-1}$. The specifications of the exec-mode API functions are all given in terms of this abstraction. For example, the postcondition of DList::new() says

```rust
struct Node<V> {
  prev: Option<PPtr<Node<V>>>,
  next: Option<PPtr<Node<V>>>,
  value: V,
}

struct DList<V> {
  #[spec] ptrs: Seq<PPtr<Node<V>>>,
  #[proof] perms: Map<nat, PermData<Node<V>>>,
  #[exec] head: Option<PPtr<Node<V>>>,
  #[exec] tail: Option<PPtr<Node<V>>>,
}

impl<V> DList<V> {
  #[spec] fn view(&self) -> Seq<V> { /* ... */ }

  #[exec] fn new() -> Self {
    ensures(|s: Self| s.well_formed(),
      && s.view().len() == 0); (k)
    /* ... */
  }

  #[exec] fn push_back(&mut self, v: V) {
    requires(old(self).well_formed());
    ensures(self.well_formed() && (l)
      equal(self.view(), old(self).view().push(v))
    );
    /* ... */
  }

  /* push_front, pop_back, pop_front similar */
}

fn main() {
  let mut t = DList::<u32>::new();
  t.push_back(2);  // 2
  t.push_back(3);  // 2, 3
  t.push_front(1); // 1, 2, 3
  let x = t.pop_back(); // returns 3
  let y = t.pop_front(); // returns 1
  let z = t.pop_front(); // returns 2
  assert(x == 3);
  assert(y == 1);
  assert(z == 2);
}
```

(a) Definition of the `DList` struct for the doubly-linked list example, along with the double-ended queue API, and example usage.

```rust
impl<V> DList<V> {
  #[spec] fn prev_of(&self, i: nat)
      -> Option<PPtr<Node<V>>> {
    if i == 0 {
      None
    } else {
      Some(self.ptrs.index(i as int - 1))
    }
  }

  #[spec] fn next_of(&self, i: nat)
      -> Option<PPtr<Node<V>>> {
    if i + 1 == self.ptrs.len() {
      None
    } else {
      Some(self.ptrs.index(i as int + 1))
    }
  }

  #[spec] fn wf_perm(&self, i: nat) -> bool { (m)
    self.perms.dom().contains(i) (n)
    && equal(self.perms.index(i).view().pptr,
      self.ptrs.index(i as int).id()) (o)
    && match self.perms.index(i).view().value {
      Some(node) => (p)
        equal(node.prev, self.prev_of(i)) &&
        equal(node.next, self.next_of(i)),
      None => false,
    }
  }

  #[spec] fn well_formed(&self) -> bool { (q)
    (if self.ptrs.len() != 0 {
      equal(self.head, Some(self.ptrs.index(0)))
    && (r)
      equal(self.tail, Some(self.ptrs.index(self.
    ptrs.len() as int - 1)))
    } else {
      equal(self.head, None) && (s)
      equal(self.tail, None)
    })
    && forall(|i: nat| imply(0 <= i && i < self.
    ptrs.len(), self.wf_perm(i))) (t)
  }
}
```

(b) Definition of `well_formed`, used internally by the `DList` implementation to prove correctness of push_back and others.

Fig. 6. Doubly-linked list example

that the list represents the empty sequence, while the postcondition of `DList::push_front(v)` says that `v` is appended to the end of the sequence. The remaining three API functions (`push_back`, `pop_front`, and `pop_back`) are similar.

The exec implementations are too involved to show here, so instead we show the definition of the spec-mode predicate `well_formed` (q), i.e., the invariant that holds on `DList<T>` and which each operation must preserve. This definition says that (r) the `head` and `tail` pointers are the first and last, respectively (unless the list is empty, in which case (s) they are both `None`). Finally, the `forall` (t) says that for each $0 \leq i < n$, `wf_perm(i)` holds; i.e., the ith permission is correct. The definition of `wf_perm(i)` (m) says that the permission is in our `perms` map (n), the permission corresponds to p_i (o), and that the `prev` and `next` fields of the node have the correct values (p).

```
struct InvCell<T> { /* ... */ }

impl InvCell<T> {
  // Well-formedness of the InvCell
  #[spec] pub fn wf(&self) -> bool;

  // Boolean predicate indicating the values
    allowed to be stored.
  #[spec] pub fn inv(&self, val: T) -> bool;

  // Construct a new InvCell, with initial value
    `val` and invariant given by `f`.
  #[exec] pub fn new(val: T, #[spec] f: impl Fn(T)
     -> bool) -> Self {
    requires(f(val));
    ensures(|cell: Self| cell.wf() && forall(|t: T
    | f(t) == cell.inv(t)));
    /* ... */
  }

  // Write to the cell and return the old value.
  #[exec] pub fn replace(&self, val: T) -> T {
    requires(self.wf() && self.inv(val)); (S)
    ensures(|old_val: T| self.inv(old_val));
    /* ... */
  }

  // Read the current value of the cell.
  #[exec] pub fn get(&self) -> T
    where T: Copy
  {
    requires(self.wf());
    ensures(|val: T| self.inv(val)); (T)
    /* ... */
  }
}
```

Fig. 7. API and specification for `InvCell<T>`.

5 SUPPORTING INTERIOR MUTABILITY

Interior mutability is a Rust pattern in which the contents (the "interior") of a datatype `X` may be modified even when it is shared via a reference type `&X`. Since `&` is supposed to be an "immutable" reference, interior mutability appears to be at odds with the core tenets of Rust's type system, and in fact interior mutability is only sound when restricted appropriately. Rust's standard library provides a handful of types with interior mutability, e.g., `Cell`, `RefCell`, `RwLock`, each of which provides a different set of restrictions and characteristics. For example, `Cell` may not be shared across threads, while `RwLock` is thread-safe but incurs all the costs of being a lock. The most flexible Rust datatype supporting interior mutability is the `UnsafeCell`, upon which the aforementioned types are implemented. Since `UnsafeCell` has no restrictions, it is—as the name implies—*not* safe in general, and implementations that use it must take great care.

Providing safe and correct versions of such types in Verus is challenging, since in our SMT encoding, values of type `&T` are always treated as immutable. Therefore, to handle any `Cell`-like datatypes, our SMT representation of `&Cell<T>` cannot include an encoding of its mutable "interior" `T`. How, then, are we able to verify programs that require reasoning about this interior?

There are two broad classes of strategies a Verus developer can use:

(1) Use linear ghost state to represent the contents of a cell, similar to the way linear ghost state represents the value pointed to by a pointer.
(2) Avoid "keeping track of" the interior value entirely. Instead, when the interior value is read, model the result as being effectively nondeterministic, potentially using invariants to restrict the set of values that can be stored in the interior.

Verus provides primitives and additional verified libraries supporting both styles, which the user can mix-and-match as needed.

The first strategy is the one used by our primitive `PCell` ("permissioned `Cell`"). In the same way that `PPtr` is our safe alternative to Rust's raw pointers, `PCell` is our safe alternative to `UnsafeCell`. `PCell` uses a ghost permission mechanism with a similar API and specification to `PPtr`, allowing us to track the interior value on the permission object.

The second strategy is exemplified by the type `InvCell` of Verus' standard library (Figure 7), which provides a `Cell`-like interface and allows the user to specify an invariant as a boolean predicate on values. Whenever they write to the cell, they must prove the written value satisfies the invariant Ⓢ, and when they read from it, they obtain an arbitrary value that they can *assume*

```
#[spec] fn expected_result() -> u64 { /* ... */ }

#[exec] fn computation() -> u64 {
  ensures(|res: u64| res == expected_result()); ⓤ
  /* ... */
}

#[spec] fn cell_value_inv(v: Option<u64>) -> bool {
  equal(v, Option::Some(expected_result()))
  || equal(v, Option::None) ⓥ
}

#[spec] fn cell_is_valid(
    cell: InvCell<Option<u64>>) -> bool {
  cell.wf()
  && forall(|v| (#[trigger] cell.inv(v) ==
    cell_value_inv(v)))
}

#[exec] fn init_cell() -> InvCell<Option<u64>> {
  ensures(|c| cell_is_valid(c));
  InvCell::new(Option::None,
    |v: Option<u64>| cell_value_inv(v)) ⓦ
}
```

```
#[exec] fn memoized_computation(
    cell: &InvCell<Option<u64>>) -> u64 {
  requires(cell_is_valid(*cell));
  ensures(|res: u64| res == expected_result());

  match cell.get() {
    Option::Some(res) => res, ⓧ
    Option::None => {
      let res = computation();
      cell.replace(Option::Some(res));
      res ⓨ
    }
  }
}

struct Client<'a> {
  cell: &'a InvCell<Option<u64>>,
}

fn main() {
  let c = init_cell();
  let client1 = Client { cell: &c };
  let client2 = Client { cell: &c };
  let x = memoized_computation(&client1.cell);
  let y = memoized_computation(&client2.cell);
  assert(x == y);
}
```

Fig. 8. Memoization example built on top of `InvCell`.

satisfies it ⓣ. We first illustrate how this can be useful in our next section, and then we discuss how `InvCell` is itself verified in terms of lower-level invariant primitives.

5.1 Verified Example: Memoized Function Calls

Memoization is an optimization technique whereby a user saves time by storing the result of a computation the first time it is invoked; on future invocations, they use the stored value. Here, we show how to memoize a function call `computation()`. In Figure 8 we use a function that takes 0 arguments for simplicity, so there is only a single value to memoize; in our supplementary materials [Lattuada et al. 2023b], we provide a slightly more complex example that memoizes a single-argument function `computation(i)`.

To set up the problem, we assume that `computation()` has a postcondition ⓤ ensuring that its result is equal to some desired (spec-mode) value, `expected_result()`. (This is similar to the setup of `fibo_impl` and `fibo` from earlier.) The aim is to construct a function `memoized_computation` that also returns `expected_result()`. To keep the problem interesting, we also insist that it be possible to share the "result store" across potentially many clients. As such, we need to use a shared reference type `&`; however, a given update invocation might need to update the result store, which requires mutability. Therefore, we need to use some form of interior mutability.

In our approach, we use an `InvCell` with a simple invariant on the data held by the cell. When initializing the cell ⓦ, we specify the data invariant as a (spec-mode) boolean predicate on the interior values; here, we set it to the function `cell_value_inv`, defined at ⓥ. The resulting property of the cell is expressed as in `cell_is_valid`. This definition says that the value is valid if and only if the stored value is either `None` (not yet computed) or contains the correct answer.

To implement `memoized_computation`, we first read from the cell; if the value we get is `Some`, then we return the value immediately ⓧ (as we can assume it satisfies the invariant we just specified). Otherwise, we perform the computation, store it in the cell, and return it ⓨ.

Finally, in `main`, we show that we can create multiple "clients," sharing a reference to the cell, and use them to call `memoized_computation`.

5.2 Invariant Primitives and InvCell Verification

Just as Rust's standard library implements Cell via UnsafeCell, in Verus we can implement and verify InvCell via our UnsafeCell-equivalent, PCell. To do this, though, we first need to introduce our *invariant primitives*.

To see what these are for, consider what happens when we try to implement InvCell<T> using UnsafeCell<T>. From the API, we know that we need to be able to write even when we only have access to a shared reference &InvCell<T>, but writing to the underlying UnsafeCell<T> requires exclusive ownership of the PermData<T> object.

Once again, we run into this problem of trying to gain exclusive ownership of something that is shared. However, we have pushed the problem one layer down—to the *ghost* layer, and this is where Verus introduces its trusted invariant primitives to escape the problem.

The two primitives are called LocalInvariant<G> and AtomicInvariant<G>.[1] Each one allows the user to store a (ghost) object G; each one allows the user to perform a ghost operation called *opening the invariant*, where they obtain temporary, exclusive ownership over the G. For example, this snippet shows how the implementation of InvCell<T>::replace temporarily gains access to the PermData<T> object:

```
impl InvCell<T> {
  pub fn replace(&self, val: T) -> T {
    requires(self.wf() && self.inv(val));
    ensures(|old_val| self.inv(old_val));

    let r;

    // Opens the invariant `&self.perm_inv` which has type `LocalInvariant<PermData<T>>`.
    // Opening the invariant is a ghost operation, and it binds to the ghost variable `perm`.
    open_local_invariant!(&self.perm_inv => perm => {
      // The code inside, however, is executable. This is where we actually perform
      // the write, using ownership of the ghost `perm` object, of type `PermData<T>`.
      r = self.pcell.replace(&mut perm, val);
    });

    r // Return the old value.
  }
}
```

The difference between the two primitives is that LocalInvariant<G> is restricted for use on a single thread: it does not implement Send or Sync, the traits Rust uses to mark thread-safety. AtomicInvariant<G> is thread-safe, and it does implement these traits: however, this comes with an additional requirement, that the invariant may only be opened for atomic operations. Since InvCell (like Rust's Cell) is for single-threaded use, we use LocalInvariant<V> here.

The reader might wonder what happens if we attempt to nest calls to the invariant-opening operation, open_local_invariant. It would certainly be unsound if we could open the same LocalInvariant<G> object twice, and obtain double-ownership of the 'T'. Indeed, Verus generates extra verification conditions to disallow such things by tracking which invariants are "open" at a given time. These verification conditions are designed to be lightweight, and they have no impact on our SMT generation for cases outside of those which use the low-level invariant APIs.

6 CONCURRENCY, USER-DEFINED LINEAR GHOST STATE, AND ATOMICS

Rust's memory safety and ownership discipline allows our verification methodology to be sound in the presence of multi-threading. However, verifying low-level code with fine-grained concurrency still requires additional techniques.

One key such technique is *user-defined ghost state*: just as Verus provides PermData to track memory ownership, the user can define their own ghost state to track elements of a custom

[1]Both these types also have additional type parameters used to specify the invariant as a boolean predicate on G.

concurrent protocol. For defining ghost state, we primarily use a technique of prior work [Hance et al. 2022], which suggests viewing user-defined ghost state as a "localized transition system." In a localized transition system, the user defines state transitions that can be expressed in terms of thread-local views of the global program state, and then proves inductive invariants on the resulting state transition system.

The result of this construction is a collection of `proof`-mode (ownership-checked) ghost types representing components of the system state, along with an API for performing operations that manipulate the ghost objects (constructing them, dropping them, or modifying them). These operations might require certain properties to hold of the ghost state, which can be proved from the inductive invariants of the transition system. Finally, the programmer can manipulate these objects like any other ghost object, e.g., putting them inside cells, invariant objects (Section 5.2), locks, atomics, or other mechanisms.

For example, we use this technique to verify a FIFO queue using a ring buffer with atomic head and tail pointers. At a very high level, we do this by first defining ghost state to represent the evolution of the FIFO state. This state includes both the head and tail pointers, and as a result, Verus gives us access to ghost objects that represent the head and tail, and we then associate these ghost objects with atomic memory using an `AtomicInvariant`.

For example, one of the transitions defined in this system (out of four total) is called `consume_start`. Its corresponding API function has the following type signature:

```
#[proof] pub fn consume_start(
  #[proof] &self, #[proof] tail: &Fifo::tail<T>, #[proof] consumer: &mut Fifo::consumer<T>
 ) -> PermData<T> { /* auto-generated by Verus ghost state machinery */ }
```

The `self` object, here, is a (ghost) metadata object that gives access to the API. The interesting parameters are the `tail`, a user ghost state object that represents the value of the tail pointer, and `consumer`, which represents the thread-local state of the consumer thread. Intuitively, this signature requires two things: first, that the client "prove" that they are the consumer by exhibiting the ghost state thread in order to perform the action. Second, that they access the tail pointer while performing the action. If the value of the tail pointers indicates that a message is waiting to be received, then the consumer thread obtains the permission to access a cell of the ring buffer, from which it can read a message, and which it relinquishes at the end of the "consume" operation. The validity of the operation (i.e., its ability to return this particular ghost object) is encoded in the correctness conditions of the transition system, and Verus requires the user to prove that these conditions hold from the inductive invariants.

Though user ghost state is usually intended for concurrent code, it is sometimes useful for single-threaded code as well. The supplementary materials [Lattuada et al. 2023b] include the following collection of examples for both single-threaded and multi-threaded code, all with user-defined ghost state:

- A concurrent FIFO queue based on a ring buffer, with head and tail pointers manipulated by atomics, as described above.
- A string interner that returns an identifier and ghost state, allowing the user to reason about the identifier as if it were the originally interned value.
- A thread-safe reader-writer lock, also implemented with atomics, which allows the user to specify an invariant on the protected data, in a similar fashion to `InvCell`.
- A (non-thread-safe) reference-counted pointer, similar to Rust's `Rc` (though without weak-pointers), which uses a `PPtr` for the heap allocation and a `PCell` for the reference counter.

Table 1. Example programs

Example	sloc spec	proof	exec	total	verif. time	Verus features
Allocator pages	5	0	18	23	0.1 s	linearity
XOR doubly-linked list	116	118	151	385	5.03 s	permissions
Fibonacci	20	16	22	58	2.19 s	
Vector	22	4	41	67	2.34 s	linearity
Interner	88	20	88	196	3.22 s	user ghost state
Memoization	23	2	43	68	2.28 s	interior mutability
`PCell` example usage	0	6	12	18	2.21 s	permissions
`PPtr` example usage	0	5	14	19	2.2 s	permissions
`InvCell`	21	9	42	72	2.24 s	permissions, `LocalInvariant`
FIFO queue	220	119	138	477	4.58 s	permissions, atomics, user ghost state
Verus Rc	119	108	97	324	3.51 s	permissions, cells, user ghost state
Verus `RwLock`	200	80	145	425	4.44 s	permissions, user ghost state

7 IMPLEMENTATION AND AVAILABILITY

We forked the Rust compiler to introduce additional hooks and typechecking rules. We then implemented Verus as a separate "driver" that links against the Rust compiler. Both our fork and Verus are open source (https://github.com/verus-lang/verus) and in use by various verification projects: the project page contains information on setting up and using the latest version of Verus. We made available a packaged artifact [Lattuada et al. 2023a] that supports running Verus as of the time of publication and reproducing the results in this paper. We are working with the Rust compiler developers to extend Rust with additional language features, such as support for ghost code, to better integrate Verus with Rust.

In Table 1 we list programs and examples we have verified using Verus. For each, we report the number of lines of `spec`, `proof`, and `exec` code, the time to verify the example, and interesting Verus features they employ. The code snippets used in the figures in the paper are extracts from these examples, which are available in full in the supplementary material [Lattuada et al. 2023b].

8 USER EXPERIENCE AND ERROR REPORTING

We discuss the Verus user experience by example. Suppose the user starts by defining an `Account` struct and an `exec` function to transfer funds between accounts.

```
6  pub struct Account { pub balance: u64 }
7
8  pub fn transfer_funds(orig: &mut Account, dest: &mut Account, amount: u64) {
9   requires([ old(orig).balance >= amount, old(dest).balance as nat + amount < u64::MAX ]);
10  ensures([ dest.balance == old(dest).balance + amount, orig.balance == old(orig).balance - amount ]);
11  orig.balance = orig.balance - amount; dest.balance = dest.balance + amount;
12 }
```

This function verifies, because Rust's type system ensures that `orig` and `dest` are not aliased. In fact, if the user accidentally aliased the two arguments when calling `transfer_funds`,

```
14 fn main() {
15  let mut acct1 = Account { balance: 20_000 };
16  transfer_funds(&mut acct1, &mut acct1, 20_000);
17  assert(acct1.balance == 10_000);
18 }
```

the user would quickly get an error from the Rust borrow checker, and Verus would not attempt to invoke Z3 to verify the invalid program, thereby allowing the user to quickly iterate by fixing the issue and re-running Verus.

```
error[E0499]: cannot borrow `acct1` as mutable more than once at a time
  --> account.rs:17:32
   |
17 |     transfer_funds(&mut acct1, &mut acct1, 20_000);
   |     -------------- ----------  ^^^^^^^^^^ second mutable borrow occurs here
   |     |              |
   |     |              first mutable borrow occurs here
   |     first borrow later used by call
```

If one wrote similar code in Dafny, using a `class` (a reference type) to represent the `Account`, they would declare the `transfer_funds` method as `method TransferFunds(orig: Accnt, dest: Accnt, amnt: nat)` with similar preconditions and postconditions. Dafny would report that the postconditions cannot be verified, which can be misleading to the developer, who has to determine that such a failure is due to potential aliasing of `orig` and `dest`; the developer would then need to add a framing condition to `transfer_funds`, `requires orig != dest`.

In response to the Rust borrow checker failure above, the user may try and fix the `main` function,

```
fn main() {
 let mut acct1 = Account { balance: 10_000 }; let mut acct2 = Account { balance: 20_000 };
 #[spec] let total_balance = acct1.balance + acct2.balance;
 transfer_funds(&mut acct1, &mut acct2, 20_000);
 assert(total_balance == acct1.balance + acct2.balance);
}
```

but inadvertently introduce a logical error, which Verus reports with precise pointers to the offending code, and the relevant context (in this case, the failing precondition on the definition of `transfer_funds`):

```
error: precondition not satisfied
  --> account.rs:18:5
   |
9  |   requires([ old(orig).balance >= amount, old(dest).balance as nat + amount < u64::MAX ]);
   |              -------------------------- failed precondition
...
18 |     transfer_funds(&mut acct1, &mut acct2, 20_000);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Adjusting the transferred amount to `10_000` would result in successful verification upon re-running Verus.

This user experience is conceptually similar to that of the Viper separation logic engine [Müller et al. 2016] and the VeriFast C and Java verification tool [Jacobs et al. 2011], with the distinction that both of these tools use a separate substructural logic to reason about memory permissions: the user has to explicitly manipulate permissions and separation logic predicates for the program data. In Rust's linear type system, memory-reasoning permissions are implicitly associated with data ownership, and manipulated by moving values, or taking references.

9 LIMITATIONS

Verus currently only supports mutable borrows (`&mut`) of data passed as arguments to a function call: mutable references in return values and explicit borrows on the right-hand side of assignments are not supported. We believe that adding more complete support with an approach similar to Creusot's is mainly a matter of syntax, interface design, and engineering.

Unlike tools that re-encode ownership properties (e.g., with separation logic in Viper [Müller et al. 2016]), Verus relies on borrow-checking rules and hence cannot reason about traditional Rust unsafe code. This may limit its applicability in applications that heavily rely on unsafe, e.g., when direct memory manipulation is required to communicate with memory-mapped devices. Section 3, Section 4, and Section 4.2 discuss encapsulations that alleviate the need for unsafe in certain contexts.

Verus is closely tied to Rust's type system, which is more limited in some ways than the dependent type systems of Coq and F*. This may preclude some more sophisticated styles of structuring proofs that are supported by Coq and F*. While the limitation on mutable borrows will be lifted in the near future, limitations tied to Rust's type system are imposed by Verus's design choices.

10 FORMALIZATION

The previous sections introduced Verus concepts by example. This section presents a small formal lambda calculus to make the concepts from the previous sections more precise. The goal of this lambda calculus is to serve as a model to demonstrate particular features and their type safety. We do not attempt to capture all of the semantics of Rust and Verus, since formalizing Rust semantics is by itself a large and challenging problem [Jung et al. 2018a; Pearce 2021; Weiss et al. 2019]. Instead, we focus on a small set of topics that are novel to Verus and are particularly relevant for type safety:

- `spec`, `proof`, and `exec` functions
- `spec`, `proof`, and `exec` variables, showing how `exec` and `proof` variables are treated linearly, while `spec` variables can capture nonlinear snapshots of data from `exec` and `proof` variables
- `spec`, `proof`, and `exec` annotations on datatype fields
- linear ghost permissions, with read-only borrowing
- ensuring termination of `spec` and `proof` code, particularly in the presence of mutation, recursive types, and higher-order features like traits or first-class functions

Since this is already a sizable list of topics, we aggressively minimize other features in our model language. First, we omit concurrency entirely. Second, we omit preconditions, postconditions, and verification condition generation, focusing instead on type safety and termination. (We believe that verification condition generation could be added in a style similar to the formalization of Linear Dafny [Li et al. 2022].) Third, our lambda calculus is a mostly-functional language that manipulates values, rather than an imperative language that mutates values stored in locations. (This contrasts with more detailed formalizations of Rust centered on locations [Pearce 2021; Weiss et al. 2019].) The model language does, however, include mutation, in the form of load and store operations that are controlled by linear ghost permissions. (The extended version of this paper [Lattuada et al. 2023c] also includes a second kind of mutation, in the form of a tiny nonlinear mutable heap.)

Since our model language is based on values rather than locations, it lacks Rust's distinction between a value (e.g. of type `int`) and a reference to that value (e.g. of type `&int` or `&mut int`). Nevertheless, we still want to capture some notion of borrowing in order to demonstrate borrowed linear ghost permissions. For this, we associate linear and shared *usages* with variable typings and expression typings. The usage "shared" represents immutable borrowing (as in `&int`), which we use for reading permissions. For simplicity, we omit mutable borrowing, instead annotating permissions with "linear".

We build these usages into the mode system, in a style similar to Linear Dafny [Li et al. 2022] (which in turn built on earlier work by Wadler's "let!" feature [Wadler 1990] and Cogent's purely functional support for borrowing [Amani et al. 2016]). We define a mode m to be spec, proof, or exec (see Figure 9), with a reflexive, transitive ordering exec $\sqsubseteq$ proof $\sqsubseteq$ spec and a least upper bound $m_1 \sqcup m_2$ that is the least m such that $m_1 \sqsubseteq m$ and $m_2 \sqsubseteq m$. We then associate a usage u with proof and exec modes, since proof and exec variables can be linear or borrowed:

$$\mu ::= \text{spec} \mid \text{proof}\, u \mid \text{exec}\, u$$

The environment $\Gamma ::= \{x_1 \mapsto \mu_1\ \tau_1, \ldots, x_n \mapsto \mu_n\ \tau_n\}$ tracks the mode, usage, and type of each variable. We refer to a binding $x \mapsto m$ linear τ as a linear binding, and we refer to $x \mapsto m$ shared τ

and $x \mapsto \text{spec } \tau$ as nonlinear bindings. We write ¡Γ to extract just the linear bindings from Γ and we write !Γ to extract just the nonlinear bindings from Γ (see Figure 10).

We write Γ_1, Γ_2 to concatenate two environments together. For writing typing rules, though, we often want to split environments in a more sophisticated way then simple concatenation. In particular, we want to split linear bindings between subexpressions while sharing nonlinear bindings among subexpressions. For this, we write $\Gamma = \Gamma_1 \# \Gamma_2$. For example, in the typing rule for adding two integers (see Figure 11), the left subexpression gets environment Γ_1 and the right subexpression gets Γ_2:

$$\frac{\mathrm{D}\,;\,\mathrm{P}_1\,;\,\Gamma_1\,;\,m \vdash e_1 : \mu\ \text{int} \qquad \mathrm{D}\,;\,\mathrm{P}_2\,;\,\Gamma_2\,;\,m \vdash e_2 : \mu\ \text{int}}{\mathrm{D}\,;\,\mathrm{P}_1 \# \mathrm{P}_2\,;\,\Gamma_1 \# \Gamma_2\,;\,m \vdash e_1 + e_2 : \mu\ \text{int}}$$

(The other environments can be ignored for now; Section 10.3 discusses D and H, and Section 10.1 discusses P and m.)

When $\Gamma = \Gamma_1 \# \Gamma_2$, all nonlinear bindings in Γ appear in both Γ_1 and Γ_2. Linear bindings, however are more subtle: a linear binding in Γ appears as-is in one of the environments (Γ_1 or Γ_2), and is demoted to mode spec in the other environment. Thus, the environment that didn't get the linear binding can still talk about the variable in specifications. For example, if Γ has a linear binding for x_2 and we split Γ into $\Gamma = \Gamma_1 \# \Gamma_2$, and Γ_1 receives the linear binding for x_2, then Γ_2 will receive a spec binding for x_2:

$\Gamma = \{x_1 \mapsto \text{exec shared } \tau,\ x_2 \mapsto \text{exec linear } \tau\}$
$\Gamma_1 = \{x_1 \mapsto \text{exec shared } \tau,\ x_2 \mapsto \text{exec linear } \tau\}$
$\Gamma_2 = \{x_1 \mapsto \text{exec shared } \tau,\ x_2 \mapsto \text{spec } \tau\}$
(See Figure 10 for a formal definition of $\Gamma_1 \# \Gamma_2$.)

Following Linear Dafny's formalization, our model language allows borrowing by temporarily viewing linear variables as shared within a lexical scope. For example, in the sequencing expression $e_1; e_2$ the first expression e_1 can view a portion of the environment Γ_b as shared, and these variables then revert to linear in e_2:

$$\frac{\begin{array}{c}\mathrm{D}\,;\,\mathrm{P}_1, \text{shared}(\mathrm{P}_b)\,;\,\Gamma_1, \text{shared}(\Gamma_b)\,;\,m \vdash e_1 : \mu_1\ \text{Unit} \\ \mathrm{D}\,;\,\mathrm{P}_2, \text{linear}(\mathrm{P}_b)\,;\,\Gamma_2, \text{linear}(\Gamma_b)\,;\,m \vdash e_2 : \mu_2\ \tau_2\end{array}}{\mathrm{D}\,;\,(\mathrm{P}_1 \# \mathrm{P}_2), \text{linear}(\mathrm{P}_b)\,;\,(\Gamma_1 \# \Gamma_2), \text{linear}(\Gamma_b)\,;\,m \vdash e_1;\ e_2 : \mu_2\ \tau_2}$$

Here, the notation $\text{linear}(\Gamma_b)$ means Γ_b with all proof/exec bindings made linear, $\text{shared}(\Gamma_b)$ means Γ_b with all proof/exec bindings made shared. (For more detail on this style of borrowing, which was inspired by Wadler's "let!" feature [Wadler 1990], see [Li et al. 2022].)

For simplicity and clarity, the model language implements a linear type system that prohibits discarding linear resources; for example, it disallows discarding permissions, and the only way to deallocate a linear `struct` is to deconstruct it with pattern matching. (Rust behaves more like an affine type system, allowing dropping of any value.) Rust includes a `Copy` trait, implemented by simple types like `bool` and `u64`, for types that are inherently nonlinear and may be freely copied. Our model language also includes a judgment $\mathrm{D}\,;\,m \vdash \tau : \text{Copy}$ (see the extended version of this paper [Lattuada et al. 2023c] for the formal definition) to indicate that a type τ may be copied or dropped, although, for simplicity, the copies and drops are explicit, using the expressions $\text{copy}(e)$ and $\text{drop}(e)$.

Even though the linear type system prohibits copying and dropping linear resources, it allows arbitrary implicit copying and dropping in specifications. It defines $\mathrm{D}\,;\,\text{spec} \vdash \tau : \text{Copy}$ to be true of all types, so that in spec mode, code can always use $\text{copy}(e)$ and $\text{drop}(e)$. Furthermore, spec variables can be implicitly copied when splitting variables among subexpressions using $\Gamma_1 \# \Gamma_2$. In particular, since environment splitting creates spec copies of linear variable bindings, spec variables

can be used to capture immutable snapshots of mutable linear resources. For example if variable x_l is bound linearly, the expression "let spec $x_s = x_l$ in e" can make a spec copy x_s of the linear variable x_l without consuming x_l. Here, e can continue to use x_l linearly while simultaneously keeping the immutable snapshot x_s. This allows specifications to talk about the past state (old snapshots) of linear resources as well as the current state, which is useful for specifications that relate old states to new states.

10.1 Permissions

Section 4.1 described how linear ghost permissions allow safe manipulation of low-level pointers. To model permissions and pointers, the model language contains a permission($i \mapsto \tau$) type representing permission to read or write a value to pointer i, which, for simplicity, is simply an integer constant. There are three operations on pointers and permissions:

- pread($i@e_p$) reads the value stored at pointer i, based on the access granted by permission e_p
- pwrite($i := e_v@e_p$) writes a new value e_v to pointer i, based on the access granted by permission e_p
- pdata(e_p) takes a spec-mode snapshot of the value currently stored at pointer i, based on the access granted by a spec-mode copy of the permission e_p

Figure 10 shows the typing rules for these operations. Each operation uses the same permission, but with a different mode. Writing requires linear access to the permission, so that no aliased views of the permission can have a stale view of the permission. Reading, on the other hand, can be performed on borrowed permissions with mode shared. Finally, pdata(e_p) uses the permission with mode spec, allowing use in specifications. For simplicity, we omit operations to deallocate permissions or allocate new permissions; we assume that all permissions are passed in to a program when the program starts and returned at the end of the program. However, the typing rule for pwrite($i := e_v@e_p$) allows the program to change the type of a permission, effectively reallocating the memory for a new type. Thus, the linear handling of permissions is crucial; if the permissions were not linear, a program could use a stale permission to read a value memory from memory with an out-of-date type, subverting type safety. Our type safety theorem (Section 10.4) ensures that this cannot happen.

While pdata(e_p) is a ghost-only operation, pread($i@e_p$) and pwrite($i := e_v@e_p$) perform run-time actions that, in an implementation, would be compiled to machine code. Since proofs and specifications are ghost code, they are not allowed to perform pread($i@e_p$) and pwrite($i := e_v@e_p$) operations. To enforce this, the typing rules include an access level m that limits what operations the code is allowed to perform. Many operations (such as integer addition) can be performed in any mode, but pread($i@e_p$) and pwrite($i := e_v@e_p$) can only be performed in exec mode:

$$\frac{\cdots}{\mathrm{D}\,;\,\mathrm{P}_1 \,\#\, \mathrm{P}_2\,;\,\Gamma_1 \,\#\, \Gamma_2\,;\,\mathrm{exec} \vdash \mathrm{pwrite}(i := e_v@e_p) : \mathrm{proof\ linear}\ \tau}$$

The bodies of exec functions are type-checked with access level exec, and can perform run-time reads and writes, while the bodies of proof and spec functions are type-checked with access level proof and spec, and therefore cannot perform run-time reads and writes.

The formal semantics of our mode language use a value permission($i \mapsto v$) to represent the storage location pointed to by pointer i, holding contents v. Notice that this storage location is just a value, so it may get passed around from expression to expression, in and out of functions, although the type system's linearity ensures that there will never be two inconsistent linear copies of a permission for the same pointer. There may, however, be many spec copies of the permission floating around that contain snapshots old permission state, and the code is allowed to execute

pdata(x_s) on these snapshots to obtain the old contents as spec values. In fact, it's important that pdata(x_s) return the contents associated with the snapshot x_s, rather than the most up-to-date value, because specifications must be deterministic: they cannot produce different values just because the state has changed.

In order to prove the type safety of our model language, we have to prove that well-typedness is preserved, including the well-typedness of permission values. For this, we use an environment P that keeps track of whether each storage location i is currently linear or borrowed (shared). We also need to type-check the snapshotted spec copies of permissions. For this, we provide a special "dead-end" rule that allows stale copies of a permission to persist as spec-only (see Figure 12); this is sound because the spec-only permission cannot be coerced back to a shared or linear permission for run-time reads and writes (hence our description of the spec copy as a "dead-end").

10.2 Functions and Lifetimes

Since our model language is a lambda calculus, it supports functions. This models both first-order functions, as shown in the examples in previous section, and higher-order features. For example, Verus supports first-class functions in specifications. Verus also supports simple traits with methods taking a `self` argument; these simple traits can encode first-class functions.

First-class functions in Rust (called closures in Rust terminology) are considerably more complicated than simply-typed lambda calculus functions, though. First, Rust distinguishes between `Fn`, which represents functions that may be called many times, and `FnOnce`, which represents functions that can only be called once. (There is also `FnMut`, which we do not model.) `FnOnce` functions may capture linear variables, while `Fn` functions cannot. We define a *callability* O ::= Once | Many to represent this distinction.

Rust first-class functions also have lifetimes associated with them, so that a function cannot outlive the variables that it captures, even if these variables are nonlinear (e.g., variables of type `&int`). Rust lifetimes are quite sophisticated, including parameterization over lifetime variables, but, for simplicity, our model language contains just two hard-coded lifetimes L ::= static | restricted. The lifetime static means that a function may be passed around freely, because it does not capture any shared variables, while the lifetime restricted means that a function may have captured shared variables, and therefore the function cannot be returned past the nearest enclosing borrowing scope. (Note that this rather strict limitation is only for the model language; the actual Verus implementation allows Rust's more sophisticated lifetime variables.) The definition function_body_context (see Figure 10) specifies exactly which variables may be captured by the body of a function definition for each of the four combinations of O and L.

Finally, Verus adds yet another dimension to functions: a mode m that represents a function being a spec function, proof function, or exec function. The typing rule for function calls $e_f\ e_a$ (Figure 12) require that the function e_f's mode be accessible according to the current access level, which means $m \sqsubseteq m_f$ if the current access level is m and e_f is a function of mode m_f.

With all of these configuration options, we can write function definitions $\lambda^m_{\mathrm{O\,L}}\ x{:}\mu\ \tau.\ e$ of type $\mathrm{Fn}^m_{\mathrm{O\,L}}\ \mu_1\ \tau_1\ \rightarrow\ \mu_2\ \tau_2$. Figure 12 shows two main rules for assigning function types to function definitions, one for non-spec functions and one for spec functions. The latter allows spec functions to capture snapshots of shared variables without worrying about lifetimes; it does not allow direct capturing of linear variables (since this would effectively discard the linear variable), but programs can always capture a linear variable indirectly by splitting off a spec copy of the linear variable from the surrounding environment and capturing the spec copy.

The language also allows spec snapshots of non-spec functions. Just as snapshots of permissions required a dead-end rule, functions also require dead-end rules, which bring a slightly annoying technicality. We could write very simple dead-end rules that just ignore the function's body

completely, and this would be sound, since a non-spec function snapshotted as a spec value can never be called, so the body doesn't matter. However, our proof of termination in Section 10.4 is based on a translation of our model language into the calculus of inductive constructions (CIC, the logic used by Coq), and for this translation we need to retain enough of the body to form a well-typed CIC term. For this, we need to relax the linearity checking in order for the retained body to remain well-typed in the model language. We write this relaxed checking using the notation $\vdash_{\text{lax}}$; the details of this are included in the extended version of this paper [Lattuada et al. 2023c].

10.3 Termination

When Verus code is compiled, all ghost code is erased (not compiled to machine code). This erasure is sound only if the ghost code always terminates with no side effects. The access level described in Section 10.1 enforces the absence of side effects. Enforcing termination, though, is more delicate because mutation and recursive types can often encode nontermination when combined with higher-order features, like traits and first-class functions.

To see how this can happen, consider the following two examples, written in OCaml. The first example creates a mutable reference that holds a function of type `unit -> unit`. It then stores a new function into the mutable reference The new function recursively calls itself by reading itself from the reference, causing an infinite loop:

```
let r: (unit -> unit) ref = ref (fun () -> ()) in
r := (fun () -> !r ());
!r ()
```

The second example passes a function to itself as an argument, using a recursive type `R` to encapsulate the function in a well-typed way. The function then calls its argument, which means it calls itself, causing an infinite loop:

```
type r = | R of (r -> unit)
let f (R x) = x (R x) in f (R f)
```

(Note: this can also be encoded directly in Rust as follows:

```
trait T { fn f(&self); }
fn rec<A: T>(x: &A) { x.f(); }
struct S {}
impl T for S { fn f(&self) { rec(self); } }
fn foo() { let s = S {}; s.f(); }
```

although this is more complicated.)

Neither of these examples would be caught by `decreases` clauses, because there are no explicit recursively-defined functions in the code.

We demonstrate that Verus can correctly prohibit these sources of nontermination by including the following features in the model language: (i) permissions (Section 10.3); (ii) a small heap consisting of one ref cell (in the extended version of the paper [Lattuada et al. 2023c]); (iii) recursive types, in the form of recursive structs, described below.

A recursive struct declaration $\mathrm{d} ::= \mathrm{S} \mapsto (m_1\ \tau_1, \ldots, m_n\ \tau_n)$ declares a struct S with n fields, each having a mode and a type. When constructing or destructing structs, the typing rules join the mode of the fields with the mode of the overall struct value using the $\sqcup$ operator (see Figure 12). This joining ensures, for example, that when reading fields from a spec snapshot of an exec struct value, the result will have mode spec even if the field mode is proof or exec.

The rules for well-formed struct declarations (included in the extended version of this paper [Lattuada et al. 2023c]) allow recursive structs (although, for simplicity, they disallow mutual recursion). These rules enforce a standard "strict positivity" restriction (used by Coq, Lean, F*, and Dafny). However, they only require strict positivity in spec and proof function types; non-positive uses are allowed in exec function types (unlike in Coq, Lean, and Dafny, where all function types are restricted).

variable	x		
integer	i	::=	$\ldots, -2, -1, 0, 1, 2, \ldots$
struct name	S		
usage	u	::=	linear | shared
mode	m	::=	spec | proof | exec
mode + usage	μ	::=	spec | proof u | exec u
callability	O	::=	Once | Many
lifetime	L	::=	static | restricted
type	τ	::=	int | Unit | permission$(i \mapsto \tau)$ | Option(τ)
		|	S | $\mathrm{Fn}^{m}_{\mathrm{O\,L}}\ \mu_1\ \tau_1 \rightarrow \mu_2\ \tau_2$
value	v	::=	i | () | permission$(i \mapsto v)$ | None(τ) | Some$(v : \tau)$
		|	$\mathrm{S}(v_1, \ldots, v_n)$ | $\lambda^{m}_{\mathrm{O\,L}}\ x : \mu\ \tau.\ e$
expression	e	::=	x | i | $e_1 + e_2$ | ()
		|	permission$(i \mapsto v)$ | pdata(e_p) | pread$(i@e_p)$ | pwrite$(i := e_v@e_p)$
		|	drop(e) | copy(e) | $e_1; e_2$ | let $m\ x = e_1$ in e_2
		|	None(τ) | Some$(e : \tau)$ | if let Some$(x) = e_1$ then e_2 else e_3
		|	$\mathrm{S}(e_1, \ldots, e_n)$ | let $\mathrm{S}(x_1, \ldots, x_n) = e_1$ in e_2
		|	$\lambda^{m}_{\mathrm{O\,L}}\ x : \mu\ \tau.\ e$ | $e_1\ e_2$
datatype decl	d	::=	$\mathrm{S} \mapsto (m_1\ \tau_1, \ldots, m_n\ \tau_n)$
datatype decls	D	::=	$\mathrm{d}_1, \ldots, \mathrm{d}_n$
permission env	P	::=	$\{i_1 \mapsto u_1, \ldots, i_n \mapsto u_n\}$
variable env	Γ	::=	$\{x_1 \mapsto \mu_1\ \tau_1, \ldots, x_n \mapsto \mu_n\ \tau_n\}$

Fig. 9. Formal Model Language Syntax

10.4 Semantics and Type Safety

The extended version of this paper [Lattuada et al. 2023c] defines evaluation rules $e \longrightarrow e'$ for an expression to take a single step to a new expression. Based on this and the typing rules, we have proven type preservation, progress, and ghost-code termination:

- Preservation: if $\vdash$ D and D ; P ; Γ ; $m \vdash e : \mu\ \tau$ and $e \longrightarrow e'$, then D ; P ; Γ ; $m \vdash e' : \mu\ \tau$
- Progress: if $\vdash$ D and D ; P ; $\emptyset$; $m \vdash e : \mu\ \tau$ and e is not a value, then there is some e' such that $e \longrightarrow e'$.
- Termination: if $m \in \{\text{spec}, \text{proof}\}$ and $\vdash$ D and D ; P ; $\emptyset$; $m \vdash e_0 : \mu\ \tau$ then there is no infinite evaluation sequence $e_0 \longrightarrow e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots$

The supplementary material [Lattuada et al. 2023b] contains proofs of these theorems. The preservation and progress proofs are straightforward. The termination proof works by translating the declarations, types, and expressions into CIC (calculus of inductive constructions) declarations and terms, and then proving that the CIC declarations and terms are well-typed and proving a simulation between the CIC reduction steps and the e evaluation steps. The translation to CIC is fairly simple: spec and proof function types are translated into corresponding CIC function types, while exec function types are simply translated into the unit type, exec functions are erased completely (translated into the unit value), and permission$(i \mapsto v)$ is translated into v.

$\text{mode\_of}(\text{spec}) = \text{spec} \qquad \text{mode\_of}(m\ u) = m$
$\text{is\_linear}(\mu) = \text{true iff } \mu = m\ \text{linear}$
$!\text{P} = \{i \mapsto \text{shared} \mid i \mapsto \text{shared} \in \text{P}\} \qquad !\Gamma = \{x \mapsto \mu\ \tau \mid x \mapsto \mu\ \tau \in \Gamma \wedge \neg\text{is\_linear}(\mu)\}$
$¡\text{P} = \{i \mapsto \text{linear} \mid i \mapsto \text{linear} \in \text{P}\} \qquad ¡\Gamma = \{x \mapsto \mu\ \tau \mid x \mapsto \mu\ \tau \in \Gamma \wedge \text{is\_linear}(\mu)\}$
$\text{linear}(\text{P}) = \{i \mapsto \text{linear} \mid i \mapsto u \in \text{P}\} \qquad \text{linear}(\Gamma) = \{x \mapsto (m\ \text{linear})\ \tau \mid x \mapsto (m\ u)\ \tau \in \Gamma\}$
$\text{shared}(\text{P}) = \{i \mapsto \text{shared} \mid i \mapsto u \in \text{P}\} \qquad \text{shared}(\Gamma) = \{x \mapsto (m\ \text{shared})\ \tau \mid x \mapsto (m\ u)\ \tau \in \Gamma\}$
$\text{spec}(\Gamma) = \{x \mapsto \text{spec}\ \tau \mid x \mapsto \mu\ \tau \in \Gamma\}$
$\text{P} = \text{P}_1\ \#\ \text{P}_2$ iff $¡\text{P} = ¡\text{P}_1, ¡\text{P}_2$ and $!\text{P} = !\text{P}_1 = !\text{P}_2$
$\Gamma = \Gamma_1\ \#\ \Gamma_2$ iff $¡\Gamma = ¡\Gamma_1, ¡\Gamma_2$ and $(!\Gamma, \text{spec}(¡\Gamma)) = (!\Gamma_1, \text{spec}(¡\Gamma_1)) = (!\Gamma_2, \text{spec}(¡\Gamma_2))$
Define $\text{is\_unrestricted}(\mu, \tau)$ to mean: $\mu \neq (m\ \text{shared})$ and $\text{mode\_of}(\mu) \vdash \tau : \text{static}$
Define $\text{is\_static}(\Gamma)$ to mean: for all $x \mapsto \mu\ \tau \in \Gamma$, $\text{lifetime\_of}(\tau) = \text{static}$
Define $\text{non\_spec\_function\_modes}(m_f, \mu_x, \mu_b, \tau_b)$ to mean: $\text{is\_unrestricted}(\mu_b, \tau_b)$ and $m_f \neq \mathit{spec}$ and $m_f \sqsubseteq \text{mode\_of}(\mu_x)$ and $m_f \sqsubseteq \text{mode\_of}(\mu_b)$
Define $\text{function\_body\_context}(\text{O}, \text{L}, \text{P}, \Gamma, \text{P}_b, \Gamma_b, u)$ to mean:

- if O = Once and L = restricted then $\text{P}_b = \text{P}$ and $\Gamma_b = \Gamma$
- if O = Many and L = restricted then $\text{P} = !\text{P}$ and $\Gamma = !\Gamma$ and $\text{P}_b = \text{P}$ and $\Gamma_b = \Gamma$
- if O = Many and L = static then $\text{P} = !\text{P}$ and $\text{P}_b = \varnothing$ and $\Gamma = !\Gamma$ and $\Gamma_b = \text{spec}(\Gamma)$
- if O = Once and L = static then $\text{P}_b = ¡\text{P}$ and $\Gamma_b = ¡\Gamma, \text{spec}(!\Gamma)$ and $\text{is\_static}(¡\Gamma)$
- if O = Once then $u = \text{linear}$

Define $\text{lifetime\_of}(\tau)$ to be:

- $\text{lifetime\_of}(\text{Fn}^{m}_{\text{O L}}\ \mu_1\ \tau_1 \rightarrow \mu_2\ \tau_2) = \text{L}$
- $\text{lifetime\_of}(\text{Option}(\tau)) = \text{lifetime\_of}(\tau)$
- $\text{lifetime\_of}(\tau) = \text{static}$ for all other τ

Fig. 10. Notation and definitions for type checking

11 RELATED WORK

Many tools for verifying Rust code exist. As far as we know, no other tool leverages Rust's borrow checker to enforce linear ghost permissions. However, in other dimensions, there is significant overlap between Verus and other projects.

Creusot [Denis et al. 2022] may be the closest tool to Verus, since it uses Rust code to express specifications and proofs, based on a macro named Pearlite. Creusot functions can be annotated as `#[logic]` or `#[predicate]` to indicate that the functions are ghost. These are similar to Verus' `spec` functions, in that they are not checked for linearity and borrowing ("Pearlite formulas are type-checked by the front-end of the Rust compiler, but they are not borrow checked"). Creusot does not have ghost code that is checked for linearity and borrowing, the way Verus' `proof` functions and `proof` variables are. Verus' SMT-LIB encoding is conceptually similar to the one produced by Creusot [Denis et al. 2022] via the Why3 [Bobot et al. 2011] prover, which requires an intermediate step: in Creusot the Rust code is first lowered into Why3's MLCFG (an ML with labelled blocks and gotos), and then Why3 encodes verification conditions for the backend solvers.

Prusti [Astrauskas et al. 2022, 2019] verifies Rust code by translating it into the Viper separation logic engine [Müller et al. 2016], effectively reverifying ownership properties enforced by Rust's borrow checker. This relatively heavyweight encoding creates larger formulas for an SMT solver, but can be used for Rust `unsafe` code that subverts Rust's borrow checking rules. By contrast, Verus relies on the memory safety enforced by Rust's borrow checker, obviating the need to use separation logic ubiquitously—instead, the user can selectively apply separation logic-style techniques (based on linear ghost permissions) only for the tricky cases that require them.

Aeneas [Ho and Protzenko 2022] verifies Rust code by translating it into a purely functional representation in F*. In this style of verification, programmers develop a proof about the functional

Well-typed expression (main rules) $\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash e : \mu\ \tau$

$$\frac{m \sqsubseteq \mathrm{mode\_of}(\mu_x)}{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma, x \mapsto \mu_x\ \tau_x\,;\,m \vdash x : \mu_x\ \tau_x} \qquad \mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma, x \mapsto m_x \text{ shared } \tau_x\,;\,m \vdash x : \mathrm{spec}\ \tau_x$$

$$\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash i : \mu\ \mathrm{int} \qquad \frac{\mathrm{D}\,;\,\mathrm{P}_1\,;\,\Gamma_1\,;\,m \vdash e_1 : \mu\ \mathrm{int} \qquad \mathrm{D}\,;\,\mathrm{P}_2\,;\,\Gamma_2\,;\,m \vdash e_2 : \mu\ \mathrm{int}}{\mathrm{D}\,;\,\mathrm{P}_1 \# \mathrm{P}_2\,;\,\Gamma_1 \# \Gamma_2\,;\,m \vdash e_1 + e_2 : \mu\ \mathrm{int}}$$

$$\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash () : \mu\ \mathrm{Unit} \qquad \frac{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash v : \mathrm{exec\ linear}\ \tau \qquad \mathrm{D}\,;\,\mathrm{exec} \vdash \tau : \mathrm{Copy}}{\mathrm{D}\,;\,!\mathrm{P}, i \mapsto u\,;\,!\Gamma\,;\,m \vdash \mathrm{permission}(i \mapsto v) : \mathrm{proof}\ u\ \mathrm{permission}(i \mapsto \tau)}$$

$$\frac{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash e : \mathrm{spec\ permission}(i \mapsto \tau)}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash \mathrm{pdata}(e) : \mathrm{spec}\ \tau} \qquad \frac{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,\mathrm{exec} \vdash e_p : \mathrm{proof\ shared\ permission}(i \mapsto \tau)}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,\mathrm{exec} \vdash \mathrm{pread}(i@e_p) : \mathrm{exec\ shared}\ \tau}$$

$$\frac{\begin{array}{c}\mathrm{D}\,;\,\mathrm{P}_1\,;\,\Gamma_1\,;\,\mathrm{exec} \vdash e_v : \mathrm{exec\ linear}\ \tau' \\ \mathrm{D}\,;\,\mathrm{P}_2\,;\,\Gamma_2\,;\,\mathrm{exec} \vdash e_p : \mathrm{proof\ linear\ permission}(i \mapsto \tau) \qquad \mathrm{D}\,;\,\mathrm{exec} \vdash \tau' : \mathrm{Copy}\end{array}}{\mathrm{D}\,;\,\mathrm{P}_1 \# \mathrm{P}_2\,;\,\Gamma_1 \# \Gamma_2\,;\,\mathrm{exec} \vdash \mathrm{pwrite}(i := e_v@e_p) : \mathrm{proof\ linear}\ \tau'}$$

$$\frac{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash e : m_e\ \mathrm{linear}\ \tau \qquad \mathrm{D}\,;\,m_e \vdash \tau : \mathrm{Copy}}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash \mathrm{drop}(e) : m_e\ \mathrm{shared}\ \tau}$$

$$\frac{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash e : m_e\ \mathrm{shared}\ \tau \qquad \mathrm{D}\,;\,m_e \vdash \tau : \mathrm{Copy}}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash \mathrm{copy}(e) : m_e\ \mathrm{linear}\ \tau}$$

$$\frac{\mathrm{D}\,;\,\mathrm{P}_1, \mathrm{shared}(\mathrm{P}_b)\,;\,\Gamma_1, \mathrm{shared}(\Gamma_b)\,;\,m \vdash e_1 : \mu_1\ \mathrm{Unit} \qquad \mathrm{D}\,;\,\mathrm{P}_2, \mathrm{linear}(\mathrm{P}_b)\,;\,\Gamma_2, \mathrm{linear}(\Gamma_b)\,;\,m \vdash e_2 : \mu_2\ \tau_2}{\mathrm{D}\,;\,(\mathrm{P}_1 \# \mathrm{P}_2), \mathrm{linear}(\mathrm{P}_b)\,;\,(\Gamma_1 \# \Gamma_2), \mathrm{linear}(\Gamma_b)\,;\,m \vdash e_1; e_2 : \mu_2\ \tau_2}$$

$$\frac{\begin{array}{c}\mathrm{D}\,;\,\mathrm{P}_1, \mathrm{shared}(\mathrm{P}_b)\,;\,\Gamma_1, \mathrm{shared}(\Gamma_b)\,;\,m \vdash e_1 : \mu_1\ \tau_1 \\ \mathrm{D}\,;\,\mathrm{P}_2, \mathrm{linear}(\mathrm{P}_b)\,;\,\Gamma_2, \mathrm{linear}(\Gamma_b), x \mapsto \mu_1\ \tau_1\,;\,m \vdash e_2 : \mu_2\ \tau_2 \\ \mathrm{is\_unrestricted}(\mu_1, \tau_1) \text{ or } (\mathrm{P}_b = \emptyset \text{ and } \Gamma_b = \emptyset) \\ \mathrm{mode\_of}(\mu_2) \vdash \tau_2 : \mathrm{static} \qquad m_1 = \mathrm{mode\_of}(\mu_1) \qquad m \sqsubseteq m_1\end{array}}{\mathrm{D}\,;\,(\mathrm{P}_1 \# \mathrm{P}_2), \mathrm{linear}(\mathrm{P}_b)\,;\,(\Gamma_1 \# \Gamma_2), \mathrm{linear}(\Gamma_b)\,;\,m \vdash \mathrm{let}\ m_1\ x = e_1\ \mathrm{in}\ e_2 : \mu_2\ \tau_2}$$

$$\frac{\mathrm{D} \vdash \tau}{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash \mathrm{None}(\tau) : \mu\ \mathrm{Option}(\tau)} \qquad \frac{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash e : \mu\ \tau}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash \mathrm{Some}(e : \tau) : \mu\ \mathrm{Option}(\tau)}$$

$$\frac{\begin{array}{c}\mathrm{D}\,;\,\mathrm{P}_1\,;\,\Gamma_1\,;\,m \vdash e_1 : \mu_1\ \mathrm{Option}(\tau_1) \\ \mathrm{D}\,;\,\mathrm{P}_b\,;\,\Gamma_b, x \mapsto \mu_1\ \tau_1\,;\,m_b \vdash e_2 : \mu_b\ \tau_b \qquad \mathrm{D}\,;\,\mathrm{P}_b\,;\,\Gamma_b\,;\,m_b \vdash e_3 : \mu_b\ \tau_b \\ m \sqsubseteq m_b \qquad (\mathrm{mode\_of}(\mu_1) \sqsubseteq m_b) \text{ or } (\mathrm{mode\_of}(\mu_1) = \mathrm{spec} \text{ and } m_b = \mathrm{proof})\end{array}}{\mathrm{D}\,;\,\mathrm{P}_1 \# \mathrm{P}_b\,;\,\Gamma_1 \# \Gamma_b\,;\,m \vdash \mathrm{if\ let\ Some}(x) = e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 : \mu_b\ \tau_b}$$

Fig. 11. Type Checking Rules

representation of executable Rust code, which is quite different from Verus' Hoare-logic style, where the programmer annotates the Rust code with preconditions, postconditions, and loop invariants.

Yanovski et al. [2021] propose a datatype called `GhostCell`, which separates data from permission in a manner similar to our `PCell` and `PermData`. The main difference is that `GhostCell` employs a polymorphic type trick to enforce that a permission may only be used with the cells to which it corresponds, while `PCell` uses a `requires` clause to enforce this, which is more flexible and allows permissions to depend on data that is not statically determined during type-checking. Furthermore, while `GhostCell` is used to enforce memory safety, to our knowledge, it has not been used to show functional correctness properties.

Well-typed expression (main rules, continued)

$$\frac{\mathrm{D} = \ldots, \mathrm{S} \mapsto (m_1\ \tau_1, \ldots, m_n\ \tau_n), \ldots \qquad \forall 1 \le i \le n,\ \mathrm{D}\,;\,\mathrm{P}_i\,;\,\Gamma_i\,;\,m \vdash e_i : (m_i \sqcup \mu)\ \tau_i}{\mathrm{D}\,;\,\mathrm{P}_1 \#\ \ldots\ \#\,\mathrm{P}_n\,;\,\Gamma_1 \#\ \ldots\ \#\,\Gamma_n\,;\,m \vdash \mathrm{S}(e_1, \ldots, e_n) : \mu\ \mathrm{S}}$$

$$\frac{\begin{array}{c}\mathrm{D} = \ldots, \mathrm{S} \mapsto (m_1\ \tau_1, \ldots, m_n\ \tau_n), \ldots \qquad \mathrm{D}\,;\,\mathrm{P}_0\,;\,\Gamma_0\,;\,m \vdash e_0 : \mu_0\ \mathrm{S} \\ \mathrm{D}\,;\,\mathrm{P}_b\,;\,\Gamma_b, x_1 \mapsto (m_1 \sqcup \mu_0)\ \tau_1, \ldots, x_n \mapsto (m_n \sqcup \mu_0)\ \tau_n\,;\,m \vdash e_b : \mu_b\ \tau_b \qquad \mathrm{mode\_of}(\mu_b) \vdash \tau_b : \mathrm{static}\end{array}}{\mathrm{D}\,;\,\mathrm{P}_0 \#\,\mathrm{P}_b\,;\,\Gamma_0 \#\,\Gamma_b\,;\,m \vdash \mathrm{let\ S}(x_1, \ldots, x_n) = e_0\ \mathrm{in}\ e_b : \mu_b\ \tau_b}$$

$$\frac{\begin{array}{c}\mathrm{D}\,;\,\mathrm{P}_b\,;\,\Gamma_b, x \mapsto \mu_x\ \tau_x\,;\,m_f \vdash e_b : \mu_b\ \tau_b \\ \mathrm{D} \vdash \tau_x \qquad \mathrm{function\_body\_context}(\mathrm{O}, \mathrm{L}, \mathrm{P}, \Gamma, \mathrm{P}_b, \Gamma_b, u) \qquad \mathrm{non\_spec\_function\_modes}(m_f, \mu_x, \mu_b, \tau_b)\end{array}}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash (\lambda^{m_f}_{\mathrm{O\,L}}\ x{:}\mu_x\ \tau_x.\ e_b) : m_f\ u\ (\mathrm{Fn}^{m_f}_{\mathrm{O\,L}}\ \mu_x\ \tau_x \to \mu_b\ \tau_b)}$$

$$\frac{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma, x \mapsto \mathrm{spec}\ \tau_x\,;\,\mathrm{spec} \vdash e_b : \mathrm{spec}\ \tau_b \qquad \mathrm{D} \vdash \tau_x}{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash (\lambda^{\mathrm{spec}}_{\mathrm{Many\ static}}\ x{:}\mathrm{spec}\ \tau_x.\ e_b) : \mathrm{spec}\ (\mathrm{Fn}^{\mathrm{spec}}_{\mathrm{Many\ static}}\ \mathrm{spec}\ \tau_x \to \mathrm{spec}\ \tau_b)}$$

$$\frac{\begin{array}{c}\mathrm{O} = \mathrm{Once} \implies \mathrm{is\_linear}(\mu_1) \qquad \mathrm{D}\,;\,\mathrm{P}_1\,;\,\Gamma_1\,;\,m \vdash e_f : \mu_1\ (\mathrm{Fn}^{m_f}_{\mathrm{O\,L}}\ \mu_a\ \tau_a \to \mu_b\ \tau_b) \\ \mathrm{D}\,;\,\mathrm{P}_2\,;\,\Gamma_2\,;\,m \vdash e_a : \mu_a\ \tau_a \qquad \mathrm{mode\_of}(\mu_1) \sqsubseteq m_f \qquad m \sqsubseteq m_f\end{array}}{\mathrm{D}\,;\,\mathrm{P}_1 \#\,\mathrm{P}_2\,;\,\Gamma_1 \#\,\Gamma_2\,;\,m \vdash e_f\ e_a : \mu_b\ \tau_b}$$

Well-typed expression (dead-end rules) $\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash e : \mu\ \tau$

$$\frac{\mathrm{spec}(\Gamma) = \mathrm{spec}(\Gamma') \qquad \mathrm{mode\_of}(\mu) = \mathrm{mode\_of}(\mu') \qquad \mathrm{D}\,;\,\mathrm{P}'\,;\,\Gamma'\,;\,m \vdash_{\mathrm{lax}} e : \mu'\ \tau}{\mathrm{D}\,;\,\mathrm{P}\,;\,\Gamma\,;\,m \vdash_{\mathrm{lax}} e : \mu\ \tau}$$

$$\frac{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash v : \mu\ \tau}{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash \mathrm{permission}(i \mapsto v) : \mathrm{spec\ permission}(i \mapsto \tau)}$$

$$\frac{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma, x \mapsto \mu_x\ \tau_x\,;\,m_f \vdash_{\mathrm{lax}} e_b : \mu_b\ \tau_b \qquad \mathrm{D} \vdash \tau_x \qquad \mathrm{non\_spec\_function\_modes}(m_f, \mu_x, \mu_b, \tau_b)}{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash (\lambda^{m_f}_{\mathrm{O\,L}}\ x{:}\mu_x\ \tau_x.\ e_b) : \mathrm{spec}\ (\mathrm{Fn}^{m_f}_{\mathrm{O\,L}}\ \mu_x\ \tau_x \to \mu_b\ \tau_b)}$$

$$\frac{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma, x \mapsto \mu_x\ \tau_x\,;\,m_f \vdash_{\mathrm{lax}} e_b : \mu_b\ \tau_b \qquad \mathrm{D} \vdash \tau_x \qquad \mathrm{non\_spec\_function\_modes}(m_f, \mu_x, \mu_b, \tau_b)}{\mathrm{D}\,;\,!\mathrm{P}\,;\,!\Gamma\,;\,m \vdash (\lambda^{m_f}_{\mathrm{Once\,L}}\ x{:}\mu_x\ \tau_x.\ e_b) : m_f\ \mathrm{shared}\ (\mathrm{Fn}^{m_f}_{\mathrm{Once\,L}}\ \mu_x\ \tau_x \to \mu_b\ \tau_b)}$$

Fig. 12. Type Checking Rules, continued

RustBelt [Jung et al. 2018a] is a verification framework that establishes a semantic model for type safety in Rust: it allows a user to verify `unsafe` code with safe APIs, i.e., prove that any well-typed, `unsafe`-free Rust program using the API will be memory safe. This makes it complementary to Verus, which relies on that memory safety, and indeed, it might be possible to use RustBelt to verify Verus' memory primitives (`PPtr` and `PCell`) and their specifications. RustBelt can also handle atomics with relaxed memory ordering [Dang et al. 2020], which Verus does not support. RustBelt is implemented in Coq, and thus proofs are written via tactics rather than by SMT.

RustBelt has also been used as part of RustHornBelt [Matsushita et al. 2022], which validates RustHorn [Matsushita et al. 2020], the encoding used by Creusot. However, RustHornBelt still requires that `unsafe` code be proved correct in Coq, while Verus provides safe, zero-cost alternatives to commonly used `unsafe` Rust features via its linear ghost state. Specifically, Verus provides `PPtr` for raw pointers and `PCell` for `UnsafeCell`, so that users can write code (which would otherwise need those `unsafe` features) within Verus itself.

Note though that while Verus supports some `unsafe` use-cases, including raw pointers, our specification for pointers is very simple, only handling pointers that point into heap allocations from the global memory allocator. A complete pointer model for Rust would support pointers to

the stack variables, cell interiors, struct fields, references, and so on, as well as handle thorny issues such as pointer provenance. By comparison, Stacked Borrows [Jung et al. 2019] is a promising operational semantics for Rust memory accesses that aims to handle all these concepts.

Separation logic [Jung et al. 2018b; O'Hearn 2007; Reynolds 2002] was one inspiration for our linear ghost permissions, although the techniques used in Verus and separation logic are quite different. In separation logic, a permission is part of the logic rather than a program-level value, and two permissions are combined together using separating conjunction. In Verus, permissions are values and two permissions are combined together by placing them in a datatype. Thus, in Verus, programmers manipulate permissions directly as data, which can require extra programmer effort, but makes generating verification conditions for an SMT solver much easier, since SMT solvers handle classical logic, not separation logic.

Another inspiration for linear ghost permissions was earlier work on using linearity in type systems to manage changing state [Crary et al. 1999; Morrisett et al. 2005; Smith et al. 2000; Zhu and Xi 2005] Alias Types [Smith et al. 2000], for example, tracks a set of constraints on the memory state, and these constraints change linearly as the memory state evolves. ATS [Zhu and Xi 2005] combines this idea, in the form of "stateful views", with reasoning about integer arithmetic via a simple dependent type system. Most similar to our approach is L3 [Morrisett et al. 2005], which treats "capabilities" (permissions) as first-class linear ghost values, as in Verus. L3 uses type variables (specifically, location variables) to connect the capabilities to pointers, whereas Verus uses SMT solving, which avoids the burden on the programmer of universally quantifying or existentially quantifying over location variables. The combination of SMT solving and Rust's automated borrow checking means that ideas from ATS and L3 are now not only possible within a mainstream language, but convenient.

Dafny [Leino 2010] and F* [Swamy et al. 2016] support ghost code and ghost variables. F* uses an effect system to distinguish ghost functions from executable functions, and has an `erased` type to represent ghost data. F* does not have a linear type system, although the F* Steel system [Fromherz et al. 2021] supports separation logic reasoning. Dafny supports `ghost` annotations on variables, similar to Verus `spec` variables, and Dafny supports `lemmas`, similar to Verus `proof` functions. Linear Dafny [Li et al. 2022] extends Dafny with linear types and borrowing, although the linearity and borrowing is less sophisticated than in Rust (for example, Linear Dafny lacks lifetime variables).

12 CONCLUSIONS

By taking advantage of Rust's linearity and borrow checking, Verus can express linear ghost permissions that aid the verification of tricky, low-level and/or concurrent code. This allows Verus to safely express code that would be unsafe in ordinary Rust, and to prove strong correctness guarantees about the code. Even for more straightforward code, Rust's type safety and control over aliasing makes verification considerably easier, allowing Verus' generation of verification conditions to treat Rust code more as functional code than as imperative code. In other words, we've found that one of the most valuable tools for verifying Rust code is Rust itself. So we conclude with a simple slogan for Verus' style of verification: ask not what verification can do for Rust — ask what Rust can do for verification.

ACKNOWLEDGMENTS

The authors would like to thank Jay Bosamiya, Nikhil Swamy, Guido Martinez, and the anonymous reviewers for their help and suggestions on the paper. Work at CMU was supported, in part, by a gift from VMware, the Alfred P. Sloan Foundation, the Intel Corporation, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab). At ETH Zurich Andrea Lattuada was supported, in part, by a gift from VMware.

REFERENCES

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2872362.2872404

Vytautas Astrauskas, Aurel Bílý, Jonás Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings (LNCS, Vol. 13260)*. Springer, 88–108. https://doi.org/10.1007/978-3-031-06773-0_5

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (LNCS, Vol. 4111)*. Springer, 364–387. https://doi.org/10.1007/11804192_17

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 53–64. https://hal.inria.fr/hal-00790310.

Alexander Borgida, John Mylopoulos, and Raymond Reiter. 1995. On the Frame Problem in Procedure Specifications. *IEEE Trans. Software Eng.* 21, 10 (1995), 785–798. https://doi.org/10.1109/32.469460

Coq Development Team. 2022. The Coq Proof Assistant. https://coq.inria.fr/.

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. https://doi.org/10.1145/292540.292564

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. https://doi.org/10.1145/3371102

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover. In *Proceedings of the Conference on Automated Deduction (CADE)*.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4603)*, Frank Pfenning (Ed.). Springer, 183–198. https://doi.org/10.1007/978-3-540-73595-3_13

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verication of Rust Programs. In *Proceedings of ICFEM 2022 - International Conference on Formal Engineering Methods (Lecture Notes in Computer Science)*. Springer Verlag, Madrid, Spain. https://doi.org/10.1007/978-3-031-17244-1_6

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473590

Google Security Blog. 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html

Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2022. *Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems.* Technical Report CMU-CyLab-22-003. CyLab, Carnegie Mellon University.

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 41–55. https://doi.org/10.1007/978-3-

642-20398-5_4

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. https://doi.org/10.1145/3371109

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* (2018). https://doi.org/10.1017/S0956796818000151

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press, USA.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023a. *Verus: Verifying Rust Programs using Linear Ghost Types – Artifact.* https://doi.org/10.5281/zenodo.7511039

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023b. *Verus: Verifying Rust Programs using Linear Ghost Types – Supplementary Material.* https://doi.org/10.5281/zenodo.7718486 The copy of record of the supplementary material is available in the ACM DL..

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023c. Verus: Verifying Rust Programs using Linear Ghost Types (extended version). (2023). https://doi.org/10.48550/ARXIV.2303.05491

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (LNCS, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Rustan Leino and Philipp Rümmer. 2010. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *Conference: Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010.* https://doi.org/978-3-642-12002-2_26

Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2022. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–28. https://doi.org/10.1145/3527313

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

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 '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 841–856. https://doi.org/10.1145/3519939.3523704

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (LNCS, Vol. 12075)*. Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2005. L3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications*. https://doi.org/10.1007/11417170_22

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (LNCS, Vol. 9583)*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1, Article 3 (apr 2021), 73 pages. https://doi.org/10.1145/3443420

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837655

Steven Vaughan-Nichols. 2022. Linus Torvalds: Rust will go into Linux 6.1. https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

Philip Wadler. 1990. Linear Types can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990.* North-Holland, 561.

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473597

Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages (PADL'05).* https://doi.org/10.1007/978-3-540-30557-6_8

Received 2022-10-28; accepted 2023-02-25