Methods for Modeling and Improving the Rust Borrow Checker

Suzan Manasreh (smanasreh6), Rohan Bafna (rbafna3),
Cameron Hoechst (choechst3), Daniel Mendes (dmendes6)

December 7, 2025

Contents

1 Problem Statement **2**

2 Prior Work **2**
2.1 Borrow Checking in Cyclone and Rust . . . 2
2.2 Refinement typing . . . 3
2.3 Stacked Borrows . . . 4

3 Methodology **5**

4 Results & Evaluation **5**
4.1 A Simple Exploration of the Existing Borrow Checker . . . 5
4.2 State Machine Model for Embedded Borrow Checking Proofs . . . 7
4.2.1 Examples . . . 7
4.3 Results . . . 8
4.4 Theoretical Formalization of ERT for Borrow Checking . . . 8

5 Conclusion **10**

Appendices **11**

A Tokenized State Machine Implementation **11**
A.0.1 Fields . . . 12
A.0.2 Invariants . . . 12
A.0.3 Transitions . . . 12

1 Problem Statement

The Rust programming language guarantees that its programs are memory-safe through a component called the borrow checker, which uses static analysis and type checking to ensure that references never live longer than the value that they point to. This approach is conservative; there are memory-safe programs in Rust that the borrow checker rejects, even though they uphold the aliasing rules.

We propose that it may be possible to implement a more advanced borrow checker by using *explicit refinement typing* [1], a technique for extending a type system by binding types with predicates which their values must satisfy and allowing the programmer to explicitly prove the well-typedness of their programs. The borrow checker must still be sound (rejecting memory-unsafe programs) but would hopefully be more complete (accepting memory-safe programs) than Rust's current borrow checker. We aim to demonstrate this possibility by implementing a minimal, Rust-like language in a proof assistant, incorporating a borrow checker that uses explicit refinement typing, proving the soundness of the borrow checker, and demonstrating that it can recognize memory-safe programs which Rust's current borrow checker would reject.

We also explore alternative methods for embedding proofs into programs and improving Rust's borrow checker. One is via state machine transition functions. This approach is based on Verus [2], a tool for verifying Rust code that's not automatic but proof-oriented. Verus has mainly been used to verify concurrent system code via tokenized state machines, but we hypothesize that it can be used to verify borrow checker rules. The state machine can enforce the borrow checker's aliasing rules via invariants that hold through all transitions, so a program with borrow checking transitions embedded into into it won't violate the borrow checker rules. It can even enforce lifetime rules without explicit lifetimes. We model the borrow checker via a tokenized state machine, show how it can be used for analysis of whether Rust programs are safe/unsafe, and prove that it's sound via Verus's verification tools. This is detailed in section 4.2.

2 Prior Work

2.1 Borrow Checking in Cyclone and Rust

Before Rust, researchers at AT&T had already investigated a similar idea to the borrow checker, i.e. region-based memory management, in a safe dialect of C called Cyclone [3]. In region-based memory management, pointer types are qualified with a "region" parameter ρ, and these region parameters support a sub-typing relation where longer-lived regions are subtypes of shorter-lived ones. If one tries to use a pointer to a finished region or writes code with ambiguous region parameters, the compiler will reject the program.

Cyclone places the burden of managing regions onto the programmer. In Cyclone, the "heap" is a region that is created at the start of the program. All "heap"-allocated values in Cyclone live forever unless one opts into a runtime garbage collector, so the heap region is more akin static memory that is either leaked or garbage-collected. To use reclaimable dynamic memory, Cyclone programmers construct regions by writing `region` r `{`s`}`, where r is the name of the region and s is a sequence of statements. Allocations that are made within a region are destroyed once execution exits the block containing s.

Rust, on the other hand, implements *ownership semantics*, where regions are defined implicitly by the lifetime of the "owner" of resource. Like in Cyclone, Rust's references are qualified with an extra parameter which is often inferred by the compiler, but because each "region" in Rust only stores one element (the corresponding owner), this extra parameter instead captures the value's lifetime – i.e. span of program points in which it is valid. If a reference's lifetime exceeds that of its owner, the compiler rejects the program, just as Cyclone rejects programs that contain pointers that refer to dead regions.

Additionally, Rust's reference types (the "immutable reference" `&T` and "mutable reference" `&mut T`) have strict aliasing rules that are enforced by checking their lifetime against the lifetime of other references. For a particular value, there can only be up to *one* `&mut T` or any number of `&T`s in-scope at any given time. If the lifetimes of a `&T` and `&mut T` collide, the borrow-checker rejects the program.

As Rust has developed since its initial release, contributors to the language have refined the scope in which a value is considered borrowed to admit a greater number of safe programs. Rust 2015 implements "lexical lifetimes," where a borrow's lifetime is defined to be everywhere it is in-scope. This was later refined into "non-lexical lifetimes" in Rust 2018, which no longer rejects a class of safe programs that construct immutable references and one mutable reference to a value, where the mutable reference is not used until after all uses

```
fn get_default_mut<'m, K, V>(map: &'m mut HashMap<K, V>, key: &K) -> &'m mut V  // m
↪  lifetime `'m` defined here
where
    K: Eq + Hash + Clone,
    V: Default,
{
    // The value is only borrowed in one of the branches (match on `Some(&mut V)`)
    // but the borrow produced by `map.get_mut(key` is considered "in scope"
    // even in the "None" branch where no reference is available
    match map.get_mut(key) { // returning this value requires that `*map` is borrowed for
    ↪  `'m`
        Some(value) => value,
        None => {
            map.insert(key.clone(), V::default());
            map.get_mut(&key).unwrap();
        }
    }
}
```

Figure 1: Most recent borrow-checker rejects a program that safely does not violate the aliasing rules. Originates from [4].

of the immutable references. While this scheme is more flexible, it still sometimes overapproximates lifetimes causing the checker to reject safe programs.

In Figure 1, the borrow-checker considers `map.get_mut(key)` borrowed for the entire body of the match across all variants. Ideally, the borrow-checker would only extend the lifetime of the borrow in the case that binds a `&mut T` (i.e. `Some(value)`) and consider it "terminated" in the case that does not return a reference (i.e. `None`). Instead, the borrow-checker simply observes that `map.get_mut(key)` has type `Option<&'m mut T>`, and because `None` has type `Option<&'m mut T>`, it considers the borrow in-scope. We hope our solution will be able to *refine* the lifetime parameter of the matched expression (which is part of its type) based on how the value is used in diverging control-flow paths.

There is existing work formalizing these borrow checking semantics for us to base our work on – [5] [6] both outline formal semantics for lifetime-checking and subtyping.

2.2 Refinement typing

As described by Jhala and Vazou [7], refinement typing allows a programming language's type system to be extended with types that contain explicit assertions about the values they hold, called *refinement types*. These assertions are logical predicates; to be well-typed, a term of a refinement type must always produce a value of the base type, such that the predicate holds. For instance, one might want to describe the type of nonempty lists of integers. This type could be constructed explicitly, or it could be formed as a refinement on the normal type of lists of integers—for instance, assuming a proper definition of `len`, the type `list(int)[l|0 < len l]` describes all lists of integers `l` such that the length of `l` is greater than 0.

Of course, for refinement types to be practical, there needs to exist a type checker that verifies that the predicates in a refinement type always hold for all of its terms. For general predicates, this is an undecidable problem (there is no algorithm that can prove or disprove any given proposition), so refinement types have classically always been *quantifier-free*, so that type checking remains automatic and decidable. To verify that a program is well-typed, the type checker generates a *verification condition* from the refinements, which certifies that the input program is well-typed, and passes the verification condition to an SMT solver to decide whether it is true or untrue.

In addition to decidable verification, the practicality of refinement types also depends on ergonomic ways to *specify* the refinements, as doing so manually for each type in a program can be labor intensive. *Liquid types* [8] are a system by which refinement constraints can be inferred automatically from a program, allowing for the programmer to benefit from the use of refinement types without having to construct the specifications themselves. Liquid types have been implemented for Rust by Lehmann et al, in the form of Flux [9], which could be useful for our needs.

However, we believe that using refinement types and SMT solvers alone may not be enough to catch every memory safe program. This is because memory safety depends heavily on the control flow of a program, which is inherently difficult to study in the general case—at least, automatically. If the programmer were to provide explicit proofs that some portions of their code were memory safe, in case the SMT solver failed to prove safety itself, and the language were able to check those proofs for correctness, more memory safe programs could be captured, potentially without having to go through all the work involved in full formal verification (which *would* imply safety as a consequence of correctness).

This is our motivation for studying *explicit refinement types*, introduced by Ghalayini and Krishnaswami [1]. Explicit refinement types are an extension to the traditional notion of refinement types—but unlike that notion, explicit refinement types have the full power of propositional logic, i.e., their propositions can have quantifiers! Of course, this comes at the cost of sometimes needing to provide explicit proofs, but conveniently, explicit refinement types also come with a language for specifying proofs, so that a full program—both the behavior and the proof—can be treated as a single object to pass to a compiler. (This can be contrasted with proof assistants, some of which can *extract* code written in their own language to a target language, e.g., Rocq can extract to OCaml; here, the code is written in the target language to begin with and no translation is done, just checking.) Like with proof assistants, the propositions given in explicit refinement types are based on the Curry-Howard correspondence, and so proving propositions is remarkably similar. However, Ghalayini and Krishnaswami state that explicit refinement types is not a "traditional dependent type theory," as the refinements can always be erased to leave a non-dependently-typed language behind; this is in fact what allows explicit refinement types to exist in the target language.

2.3 Stacked Borrows

Prior aliasing models use stacks (Stacked Borrows (2020) [10]) and trees (Tree Borrows (2025) [11]). However, VerusSync doesn't support stacks or trees, so we used a simpler map approach for tracking state. The authors developed these aliasing models because Rust compilers make use of aliasing models for compiler optimizations. We won't explore how our aliasing model can enable compiler optimizations, but it's a possible extension.

We based our model around the same borrow checker principles that Stacked Borrows uses. The two principles are:

Definition 2.1 (Exclusion Principle) *Data can either be mutated through one reference, or it can be immutably shared amongst many parties—but not both at the same time.*

Definition 2.2 (Stack Principle) *Every use of the reference (and everything derived from it) must occur before the next use of the referent (after the reference got created).*

The exclusion principle is also referred to as the aliasing (aliasing xor mutability) rule in Rust literature. The stack principle is enforced via lifetimes in Rust's borrow checker. However, stacked borrows differs from the static analysis Rust's borrow checker does because it doesn't assign variables lifetimes. Instead, it uses dynamic analysis to ensure the stack principle holds based on program execution. There are no explicit lifetimes for references.

For example, a stack can be used to detect that this re-borrowing program is invalid:

```
1  let mut local = 0;
2  let x = & mut local ;
3  let y = & mut *x; // Reborrow x to y.
4  *x = 1; // Use x again .
5  *y = 2; // Error ! y used after x got used
```

The stack of references for the location of local after line 3 executes is: (x_{id}, y_{id}). On line 4, x gets used, so we need x_{id} at the top of the stack. This pops y_{id} off the stack. When y gets used again, we're not able to find its id on the stack, and this forms an invalid program. The time complexity isn't discussed in Stacked Borrows (2020) [10], but we believe it's $O(n)$ worst case where $n =$ number of total references.

This dynamic analysis with no specific lifetime computations is the same idea we want to use for our state machine borrows in section 4.2. We won't be doing any dynamic analysis because we did not get all the way to extending the Rust interpreter. However, the ghost state transitions are similar because they execute dynamically. They just have to be defined by the user for now.

3 Methodology

In order to implement and validate our approach, our original plan was to represent a small subset of the Rust language in a proof assistant, such as Lean, and implement the borrow-checking rules as a part of a library. Success for the project would be to find at least one program that Rust's borrow checker rejects but our system accepts, and to show that our system is sound. We also define success as an improvement on Rust's borrow checker or the various approaches like Stacked Borrows (2020) in terms of time complexity or other factors.

These were the specific steps we planned to achieve this:

1. Decide on a subset of Rust to implement.
2. Formalize its operational semantics.
3. Study a methodology (such as refinement types) to determine how it can improve borrow checking.
4. Implement the semantics using this methodology (i.e. a Rust-subset augmented with refinement types in Lean).
5. Prove progress and preservation for our language or induction.
6. Prove the soundness of our new semantics (i.e. that it never accepts a memory-unsafe program).
7. Construct a memory safe example program that Rust rejects which our borrow checker accepts.

4 Results & Evaluation

Our goals were quite ambitious in this project. Even ”simplified” formalizations such as RustBelt [5] are quite complicated. As such, we achieved only the goal of formalizing the semantics of a simple borrow checker for some methods. We explored three possible directions of increasing complexity to both vet our original idea as well as explore the possibility of a simpler solution that achieves the same goal. Our most complex solution is a lambda calculus with refinement types formalized on paper and in Lean, where we implement a subset of the borrow-checking rules as a part of its type system. Our simplest is a python interpreter implementation of the borrow checker, and our in-between is a state machine model for the borrow checker.

4.1 A Simple Exploration of the Existing Borrow Checker

As an initial step in exploring the borrow checker and providing a concrete way to experiment with simple ideas and identify existing limitations, we developed a Python-based interpreter for a simple, borrow-checked imperative language. It can be found in the directory “python-simple-interpreter”[1]. If we had successfully created and proved correctness for a new type system, this could become the basis of an implementation. The interpreter consists of two phases: the first performs type and borrow checks to detect correctness violations, and then secondly the program is actually executed.

It is sophisticated enough to include arrays, function calls, and references. It enforces aliasing XOR mutability semantics for references as can be seen with the final two test cases of the artifact. Here is an example program it would accept.

[1]The full code in this directory can be found on GitHub: `https://github.com/suzm10/6390-project`

```
test = [
    ['main', [], 'returns', 'i32', [
        ('assign', 'x', 'i32', '5'),
        ('assign', 'x_ref1', '&shared i32', '&shared x'),
        ('assign', 'x_ref2', '&shared i32', '&shared x'),
        ('assign', 'y', 'i32', '10'),
        ('assign', 'z', 'i32', ('call', 'max', ['x', 'y'])),
        ('print', 'z'),
        ('return', '0')
        ]
    ],
    ['max', ['i32', 'i32'], 'returns', 'i32', [
        ('assign', 'a', 'i32', 'arg0'),
        ('assign', 'b', 'i32', 'arg1'),
        ('if', 'a > b', [
            ('return', 'a')
        ], [
            ('return', 'b')
        ])
    ]]
]
MyInterpreter(test, ('call', 'main', []))
```

With this, we can explore several correct programs that are rejected by the borrow checker, and we found this interesting program.

```
['main', [], 'returns', 'i32', [
    ('assign', 'arr', '[i32]', '[1,2,3,4,5]'),
    ('assign', 'z1', '&mut i32', '&mut arr[0]'),
    ('assign', 'z2', '&mut i32', '&mut arr[1]'),
    ('return', '2')
]]
```

This fails to compile for an aliasing violation, but we know that is actually correct. We can find that the code responsible treats a borrow of an array at an index counts as a borrow for the entire array, and this is what Rust does. Since any variable may have one mutable borrow, we can only mutably borrow at one index. **By having modified our borrow checker to track actively borrowed indices, we can safely accept a wider set of programs where borrows at an index are known not to overlap in the array**. For details, see the `add_borrow` function in `borrow_check_{old,new}.py` from the artifact.

This suggests a further, more general improvement. Instead of enforcing ”a mutable borrow of an array at an index is disallowed if there are any existing mutable borrows” we augment the rule to ”a mutable borrow of an array at an index is disallowed if it cannot be proven to not overlap with existing mutable borrows”. In order to make an attempt at this idea, we theorized the ability to attach proofs to a multi-borrow statement proving that the indices are non-overlapping. This would be useful in a program where, for example, we test that $x > 3$ before borrowing at index 3 and x.

To achieve this in full generality would require implementing Hoare logic semantics for our language so that ”if” conditions like $x > 3$ can become part of the context of true propositions at the program point where we want to attach a proof. Instead of implementing Hoare logic, we considered embedding our language and its associated proof objects in Lean so that we can use its rich selection of tactics (e.g. `omega`) to automatically prove cases like the example program above. We did not achieve this goal, but to successfully implemented a modified type system with refinement types, then we would need to have faculties for reasoning about these properties.

4.2 State Machine Model for Embedded Borrow Checking Proofs

Our original goal was to build a verification tool. As part of this, we had to read about many existing verification tools such as Thrust [12] and RustHornBelt [13]. One of these includes Verus [2], a proof-oriented manual verification tool for Rust. Since it's built on top of Rust, it didn't look perfectly suited to our problem of being able to verify Rust programs that the Rust borrow checker doesn't allow but are syntactically correct. Therefore, in this section, we reduce the problem to just modeling the basic Rust borrow checker rules, but we hope this can form a basis for future work on dynamic analysis of Rust programs.

Verus specifically provides a tool called VerusSync which allows users to define a ghost state using a ghost state description language. Specifically, the user can define the ghost state they want, the transitions they want to perform with it, and prove invariants and other well-formedness conditions on it [2]. Therefore, we can write regular Rust programs and prove that they're safe using the aliasing model in ghost state. This section mainly focuses on defining this borrow checker state machine and the invariants/transitions on it needed to represent valid programs.

In order to embed proofs into Rust code, we first need a tokenized state machine model of the borrow checker. For the tokenized state machine, we plan to use VerusSync's map strategy, so we can have maps for whether data is mutable or immutable, counts for mutable references, counts for immutable references, and a mapping of references to referents.

For tokenized state machines, Verus produces a set of token types such that any state of the system represents a collection of token types [2]. The transitions in the state machine can add/remove tokens in the state and require them in the state.

The tokenized state machine consists of fields, invariants, transitions, and inductive proofs on those transitions. The tokens come from the fields. For each field, Verus generates a token type depending on the sharding strategy. The strategy is just a data structure such as variable, option, or map that determines how tokens are organized. For the map strategy that we use, there's a token for each key value pair.

We describe the implementation of our tokenized state machine in Appendix A. One can also read the code on GitHub to understand it[2]. In summary though, we had a mutable reference counter map `mut_map`, immutable reference counter map `imm_map`, map of references to referents `ref_map`, and a map of data to a mutability boolean `data_map`. Our invariants include non-zero immutable references, 0 or 1 mutable references, and the exclusion principle for exclusively shared immutable references or exclusively single mutable references. The transitions are demonstrated through our examples.

We created this implementation and had Verus can check the soundness of our state machine automatically (all invariants hold through all transitions, etc.). If this is successful, in theory, we can embed proofs into Rust code using our ghost state. However, the embedded proofs are lengthy and hard to understand, so for the report, we plan to explain the ghost state transitions through comments on Rust code.

4.2.1 Examples

Now that we have our state machine defined, let's see how to apply it to some Rust examples in theory. First, let's look at a re-borrow example that passes Rust's borrow checker. We show some key field updates that happen below each line from the transitions.

```rust
let mut local = 0; // add_new_data(local, true)
// data_map += (local, true), imm_map += (local, 0), mut_map += (local, 0)
let x = &mut local; // add_mut_ref(x, local)
// ref_map += (x, local); data_map += (x, true); mut_map[local] = 1
let y = &mut *x; // add_mut_ref(y, x)
// ref_map += (y, x); data_map += (y, true); mut_map[x] = 1
*y = 2; // drop_mut_ref(y)
// mut_map[x] = 0
*x = 1; // drop_mut_ref(x)
// mut_map[local] = 0
```

[2]The full code for this section is available on GitHub at: `https://github.com/suzm10/verus-borrow-checker-model/blob/main/src/borrow.rs`

All our invariants hold on the data structures through the transitions as verified by Verus. Therefore, this program is a valid re-borrow case. We can also detect invalid re-borrow cases like the one below through our `mutate_referent` transition. This transition checks for outstanding mutable or immutable references on the referent and fails if so. We know that catching a borrow checker violation through a transition is not ideal, but this is what we have for now.

```
let mut local = 0; // add_new_data(local, true)
// data_map += (local, true), imm_map += (local, 0), mut_map += (local, 0)
let x = &mut local; // add_mut_ref(x, local)
// ref_map += (x, local); data_map += (x, true); mut_map[local] = 1
let y = &mut *x; // add_mut_ref(y, x)
// ref_map += (y, x); data_map += (y, true); mut_map[x] = 1
*x = 1; // mutate_referent(x); drop_mut_ref(x)
// mut_map[x] > 0 ? invalid mutation
*y = 2;
```

An approach without `mutate_referent` would require Verus to support loops or recursion. The problem occurs because we have to call `drop_mut_ref(y)` after we call `drop_mut_ref(x)` in this example, and we need to check that `x` is still valid data to refer to. We could do this very easily in our code for `drop_mut_ref` using `ref_map`, but the problem occurs when we have re-borrows at a depth greater than 1. We don't think Verus supports recursion or loops, so we can't recurse down the `ref_map` to check that all data is valid.

The downside of the recursion on map keys approach is that it makes our approach very similar to stacked borrows. It would have the same $O(n)$ time complexity for very nested borrows. Our current approach with `mutate_referent` is an $O(1)$ check to catch the stack principle violation.

4.3 Results

When we run Verus on our tokenized state machine model, it prints:

```
verification results:: 13 verified, 0 errors
```

Therefore, we've verified our state machine is sound. It preserves its invariants through all transitions, and all the data structure manipulations are consistent with what's required for the map sharding strategy.

We haven't verified my model works for all Rust programs since this is an embedded proof assistant, but it could be implemented as an interpreter and then checked that way. We don't provide an example of an embedded proof for a real Rust program either in our code sample because we had some technical difficulties there too. However, we achieved the theory part.

4.4 Theoretical Formalization of ERT for Borrow Checking

To explore the viability of refinement types for borrow-checking, we took an approach based around *prophecy variables*, a type of construct that enables reasoning about the change in the state of a program variable over time. Our usage of prophecy variables is modeled after Thrust [12], a refinement type system to verify properties about Rust programs. Thrust uses prophecy variables to model the state of mutable references over the run of a program. In Thrust, creating a mutable reference produces a pair of variables, denoted the *initial value* and the *prophecy*, that can be used in refinements. The initial value denotes the value of the referent at the time the mutable reference is created, and the prophecy denotes the value at the time the mutable reference is destroyed, which occurs either through assigning to the reference or manually dropping it. For illustration, here is an example from the Thrust paper showing the flow of prophecy variables with re-borrowing:

```
let mut x = 2;
// x : ⟨2⟩
let m = &mut *x; // prophecy l1
// x : ⟨l1⟩, m : ⟨2, l1⟩
let n = &mut *m; // prophecy l2
// x : ⟨l1⟩, m : ⟨l2, l1⟩, n : ⟨2, l2⟩
*n -= 1; // assume l2 = 2 - 1
*m -= 1; // assume l1 = l2 - 1
assert!(x == 0);
```

To ensure that prophecy variables properly track the state of their referents, Thrust makes the assumption that if a mutable reference to a particular variable exists, then no other references to that variable may exist (this is sometimes called the "aliasing-xor-mutability" rule) and that mutable references are dropped once assigned to; Thrust justifies the former assumption as it is guaranteed by Rust's borrow checker, and the latter assumption is made for the sake of simplicity.

We believe that the problem of determining whether a pointer is used after it is dropped can be reduced to the problem of tracking all changes to mutable references. For an informal argument, consider any language with typed references that can be dropped, and then consider a modification of that language where references of type T are translated to references of type $1+T$, i.e., the coproduct of the unit type and T, where dropping a reference is translated to replacing the value of the reference with the unit value. Any program in the original language can be modeled as a program in the modified language, which is memory safe by construction, and pointer drops in the original language are exposed as changes in the state of the references in the modified language. Thus, instead of tackling use-after-free bugs directly, we attempted to use explicit refinement types to track all mutable state changes in a language *without* dropping pointers.

Our approach is, in essence, an attempt to combine of the lambda calculus with explicit refinement types (λ_{ERT}) introduced in [1] with mutable references as described in *Types and Programming Languages* [14]. We were inspired by Thrust's approach; however, as we developed the approach, we found some fundamental issues with applying a prophecy-based refinement type system like Thrust to checking memory safety. Namely, because Thrust assumes aliasing-xor-mutability and that mutable references are dropped when assigned to, many memory-safe programs—and perhaps, any memory-safe program that violates the rules of the Rust borrow checker—would still not be possible to express within this system, defeating the purpose. Unfortunately, this means that the refinement type system presented here may need to be rethought from the ground up to achieve our original goals; we present it nevertheless.

Starting from λ_{ERT}, we add reference types $\text{Ref}\,T$, a reference operator $\text{ref}\,t$, a dereference operator $!r$, an assignment operator $r := t$, and location values, formalized as they are in [14]. (The assignment rule is given later; we omit the rules for location values and store typings, which are needed for the operational semantics of STLC with references, for the sake of simplicity.)

$$\textsc{T-Ref}\quad \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{ref}\,t : \mathsf{Ref}\,T} \qquad\qquad \textsc{T-Deref}\quad \frac{\Gamma \vdash t : \mathsf{Ref}\,T}{\Gamma \vdash\, !t : T}$$

We fix evaluation order to applicative, which, because applicative evaluation order is deterministic, allows sequencing instructions through evaluation order as demonstrated in [14]. For instance, the sequencing operator $M; N$ can be desugared to the syntax $(\lambda x : \text{Unit} \,.\, N)M$.

To add prophecy variables to explicit refinement types, we make several changes to the type system of λ_{ERT}.

Lambda abstractions which take as their argument references introduce a *ghost variable*, which represents the final value of the reference at the end of evaluation of the function, to the context, allowing refinements in the output type to refer to the final value of the reference. Ghost variables, and ghost terms more broadly, are a construct used in explicit refinement types [1] that enforce *proof irrelevance*; they are terms which the type system guarantees cannot show up in normal program code, but can show up in proofs of propositions. Separating ghost terms from normal terms in λ_{ERT} allows refinements to be fully erased in a program, turning it into a normal STLC program, which Ghalayini et al. use to guarantee that explicit refinement types add no extra computational complexity to existing programs. We co-opt ghost variables to ensure that prophecies can be used in refinements without allowing programs to observe prophecies when they are run, which would make the language impossible to execute.

$$\frac{\Gamma, x : \mathrm{Ref}\, A, \|{*x} : A\| \vdash e : B, \quad \Gamma, x : \mathrm{Ref}\, A, \|{*x} : A\| \vdash B\,\mathrm{typ}}{\Gamma \vdash \lambda x : \mathrm{Ref}\, A.\, e : B}\;\textsc{T-Abs}$$

Assignments to references return a refinement type, where the base type is Unit as in [14] and the refinement enforces that the value of the prophecy equals the value assigned, roughly following Thrust's T_Assign rule.

$$\frac{\Gamma \vdash t_1 : \mathrm{Ref}\, T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 := t_2 : \{- : \mathit{Unit}, *t_1 == t_2\}}\;\textsc{T-Assign}$$

Unfortunately, this rule leads to a problem that may affect correctness in subtle ways: it assumes that only one assignment to a reference may ever happen. Without resorting to Rust's own borrow checker, it is not easy to verify that this condition holds; consider the following pseudo-program as an example.

$$\lambda x : \mathrm{Ref\,Nat}.\, \lambda y : \mathrm{Unit}.\, x := *x + 1; *x$$

This function takes a reference x and returns a *closure* that, when executed, increments the value of $*x$ and returns it. This function could be used in a program that calls the closure multiple times, resulting multiple writes to x. Rust solves this problem by having three separate kinds of functions (Fn, FnOnce, and FnMut) that carry borrow-checking information and determine whether a function can be called multiple times. However, incorporating this system into our own would be akin to relying on the Rust borrow checker, which would decrease its expressive power. Thrust sidesteps this problem, requiring that higher-order functions don't contain free variables, but suggests an approach using existential quantification, which we have not had the opportunity to study in depth.

Another potential issue in our approach is that refinements can refer to any term that exists in the base language; with references, these terms may contain side effects through assignment, and it is not exactly clear if or how side effects should be restricted in refinements. One option is to allow side effects, but limit their scope to a separate program store so that refinements don't affect the execution of a program. Another option is to somehow disallow side effects in refinements entirely, but we're not sure how this could be done.

5 Conclusion

We explored several approaches to either improve or increase the flexibility of borrow-checking compared to Rust. Although our project aimed to synthesize all our work to produce a new semantics with an example accept/reject program, we achieved more modest results - several empirical explorations (i.e. code) of the borrow checker and in-progress theoretical work defining a new type system using explicit refinement types to track mutable state changes in a language with references. We have a simpler example of an accept/reject program (Section 4.1) not based on an ERT-based type system. We had begun work on formalizing the ERT system in Lean; however, with important theoretical issues unresolved, we weren't able to give a complete formal description of our ERT-based type system. With more time, we could complete and synthesize our results into a complete implementation of a proof-of-concept language which is able to use explicit refinement types to guarantee memory safety without relying on Rust's borrow checking rules.

References

[1] J. E. Ghalayini and N. Krishnaswami, "Explicit refinement types," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, Aug. 2023. DOI: `10.1145/3607837`. [Online]. Available: `https://doi.org/10.1145/3607837`.

[2] T. Hance, "Verifying Concurrent Systems Code," Ph.D. dissertation, Carnegie Mellon University, 2024.

[3] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02, Berlin, Germany: Association for Computing Machinery, 2002, 282–293, ISBN: 1581134630. DOI: 10.1145/512529.512563. [Online]. Available: https://doi.org/10.1145/512529.512563.

[4] https://users.rust-lang.org/t/polonius-is-more-ergonomic-than-nll-but-why-hasn-t-it-been-integrated-into-the-stable-or-nightly-channels-yet/129749.

[5] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158154. [Online]. Available: https://doi.org/10.1145/3158154.

[6] A. Weiss, O. Gierczak, D. Patterson, and A. Ahmed, *Oxide: The essence of rust*, 2021. arXiv: 1903.00982 [cs.PL]. [Online]. Available: https://arxiv.org/abs/1903.00982.

[7] R. Jhala and N. Vazou, *Refinement types: A tutorial*, 2020. arXiv: 2010.07763 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2010.07763.

[8] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," *SIGPLAN Not.*, vol. 43, no. 6, 159–169, Jun. 2008, ISSN: 0362-1340. DOI: 10.1145/1379022.1375602. [Online]. Available: https://doi.org/10.1145/1379022.1375602.

[9] N. Lehmann, A. T. Geller, N. Vazou, and R. Jhala, "Flux: Liquid types for rust," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. DOI: 10.1145/3591283. [Online]. Available: https://doi.org/10.1145/3591283.

[10] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020, ISSN: 2475-1421. DOI: 10.1145/3371109. Accessed: Dec. 6, 2025.

[11] N. Villani, J. Hostert, D. Dreyer, and R. Jung, "Tree Borrows," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1019–1042, Jun. 2025, ISSN: 2475-1421. DOI: 10.1145/3735592. Accessed: Dec. 7, 2025.

[12] H. Ogawa, T. Sekiyama, and H. Unno, "Thrust: A prophecy-based refinement type system for rust," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025. DOI: 10.1145/3729333. [Online]. Available: https://doi.org/10.1145/3729333.

[13] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer, "RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, San Diego CA USA: ACM, Jun. 2022, pp. 841–856, ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523704. Accessed: Dec. 7, 2025.

[14] B. C. Pierce, *Types and Programming Languages.* MIT Press, 2002.

[15] Verus Lang, *Verus examples*, Dec. 7, 2025. [Online]. Available: https://github.com/verus-lang/verus/blob/8e79b21bfcb78f3a3db69b99be6d697464218166/examples/state_machines/flat_combine.rs#L49.

Appendices

A Tokenized State Machine Implementation

In this section, we describe how we implemented our tokenized state machine, the invariants defined on it to uphold the exclusion and stack principles, and the transitions we use to model the flow of execution of Rust code. Then, we show it passed verification and provide an initial time complexity analysis and comparison to stacked borrows.[3].

[3]The full code for this section is available on GitHub at: https://github.com/suzm10/verus-borrow-checker-model/blob/main/src/borrow.rs

A.0.1 Fields

We have three fields declared with the map sharding strategy:

1. `data_map: Map<String, bool>` which maps the original data (no references, just a regular `int` or other word in memory) to whether it's mutable (`bool == true`) or immutable (`bool == false`).
2. `imm_map: Map<String, int>` which counts the number of immutable references to a data item.
3. `mut_map: Map<String, int>` which counts the number of mutable references to a data item.
4. `ref_map: Map<String, String>` which maps the reference to the the data item it references.

These are all initialized as empty maps when the state machine is instantiated.

A.0.2 Invariants

We have four invariants in our state machine. We followed Verus's examples to define these [15]:

1. **Non-zero immutable references**:

```
forall |data: String|
    self.imm_map.dom().contains(data) ==>
        (self.imm_map[data] >= 0)
```

2. **Single mutable references**:

```
forall |data: String|
    self.mut_map.dom().contains(data) ==>
        (self.mut_map[data] == 0 || self.mut_map[data] == 1)
```

3. **Exclusion Principle**:

```
forall |data: String|
    self.data_map.dom().contains(data) ==>
        (self.mut_map[data] == 1 ==> self.imm_map[data] == 0)
```

4. **Exclusion Principle Reversed**:

```
forall |data: String|
    self.data_map.dom().contains(data) ==>
        (self.imm_map[data] > 0 ==> self.mut_map[data] == 0)
```

We might not need the non-zero immutable references invariant if we define our `imm_map` as `Map<String, nat>`, but we couldn't get Verus to verify our state machine passes due to later `imm_map` value decrements.

A.0.3 Transitions

We have transitions for adding to each map, dropping references, and mutating referents. We needed the mutate referents transition to uphold the stack principle because it checks that there are no existing references in the `imm_map` and `mut_map`. Therefore, the referent mutation will only happen after all its immutable and mutable references are not going to be used anymore.

1. `add_new_data(data: String, mutability: bool)` adds the key-value pair `(data, mutability)` to the `data_map`. Adds `(data, 0)` to both the `imm_map` and `mut_map` to initialize mutable and immutable references to 0.
2. `add_imm_ref(reference: String, data: String)` checks that the `data_map` contains the key `data`. Then, it checks that the `mut_map` has 0 mutable references for that data. Then, it adds the pair `(reference, data)` to the `ref_map` and `(reference, false)` to the `data_map`. Finally, it increments the number of current immutable references for that data item in `imm_map`.

3. `add_mut_ref(reference: String, data: String)` checks that the `data_map` contains the key `data`, and its value is set to true for mutability. Then, we check that no immutable references exist and add the pair `(reference, data)` to the `ref_map`. We also add the pair `(reference, true)` to the `data_map`. Finally, we map the number of mutable references for that data item to 1.

4. `drop_imm_ref(reference: String)` gets the data item corresponding to the reference using the `ref_map`. Then, it checks that the data item is in the `data_map`. Then, it decrements the `imm_ref` counter for that data item. It requires that the current value is greater than 0 before decrementing, so we keep non-negative immutable references.

5. `drop_mut_ref(reference: String)` also gets the data item corresponding to the reference using the `ref_map`. Then, it checks that the data item is in the `data_map` with mutability set to true. Then, it sets the counter for that item to 0. There's no need to decrement since a mutable reference can only have a count of 0 or 1.

6. `mutate_referent(data: String)` requires that mutability be set to true for that data item and that the number of mutable and immutable references for that item is also 0.

Now, we've finally defined our tokenized state machine. We also have inductive proofs for all our transitions, so if we apply transitions multiple times, Verus checks that we end up in valid states.