View Types in Rust

Sasha Pak
Australian National University
Canberra, Australia
sasha.pak@anu.edu.au

Richard Willie
National University of Singapore
Singapore, Singapore
richardw@u.nus.edu

Umang Mathur
National University of Singapore
Singapore, Singapore
umathur@nus.edu.sg

Fabian Muehlboeck
Australian National University
Canberra, Australia
fabian.Muehlboeck@anu.edu.au

Alex Potanin
Australian National University
Canberra, Australia
alex.potanin@anu.edu.au

Abstract

Rust's ownership-based type system ensures safety but often rejects intuitive and safe code, leading to workarounds. View types are a promising language extension aiming to increase flexibility and expressiveness in function calls. We outline the motivation, challenges, and research directions for integrating view types into Rust.

CCS Concepts: • **Software and its engineering** → **Language types**; **Software notations and tools**; *Software verification.*

Keywords: Rust, Partial Borrows, View Types, Ownership, Borrow Checker

ACM Reference Format:
Sasha Pak, Richard Willie, Umang Mathur, Fabian Muehlboeck, and Alex Potanin. 2025. View Types in Rust. In *Companion Proceedings of the 2025 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3758316.3765481

1 Introduction

Rust is a systems programming language that combines safety guarantees with performance. Unlike many other languages, Rust achieves memory safety without a garbage collector, relying instead on its type system and a static analysis phase called *borrow checking*. However, developers often find that their intuitive designs do not borrow-check, forcing them to restructure code in strange ways. This effect is widespread: a GitHub search reveals over 2000 comments similar to "just to make the borrow checker happy" [7]. In this work, we focus on the problem of interprocedural (or partial) borrowing in Rust and explore view types as a potential solution.

2 Partial Borrowing and View Types

Consider the following Rust program, which attempts to assign unique IDs to each item in a list using a helper method:

```
1  struct S { next_id: usize, data: Vec<Item> }
2  impl S {
3      fn new_id(&mut self) -> usize {
4          let i = self.next_id; self.next_id += 1; i
5      }
6      fn assign_ids(&mut self) {
7          for item in &mut self.data {
8              let id = self.new_id();
9              // ------^^^^-- Cannot borrow *self
10             register_item(item, id);
11         }
12     }
13 }
```

This code does not compile due to conflicting mutable and immutable borrows of `self`. During borrow checking, Rust treats functions calls as opaque and assumes that `new_id()` arbitrarily mutates all fields of `S`, which conflicts with ongoing iteration over `data`. As a result, developers are forced to restructure their code or use workarounds.

View types address this limitation [5]. They enrich references with information about exactly which field of a struct they can access, and whether each field is accessed mutably or immutably. For example, the type `&{mut next_id} S` denotes a reference to a struct `S` that grants mutable access only to the `next_id` field, leaving other fields available for separate borrowing. In general, one can specify access to multiple fields, with mutability as needed.

3 Motivating Examples

Many real-world Rust projects could benefit from partial borrowing. We highlight a few representative cases:

- Moonfire-NVR [6]: Groups mutated fields into a local struct to avoid double self-borrowing. With view types this grouping would become redundant.
- Catalyst-Core [6]: Clones a field just to satisfy the borrow checker, incurring unnecessary overhead. With view types, direct iteration would be possible.

SPLASH Companion '25, Singapore, Singapore

ACM ISBN 979-8-4007-2141-0/25/10
https://doi.org/10.1145/3758316.3765481

```
self.voteplan_managers = self
    .voteplan_managers
    .clone()
    .into_iter() ...
```

- Jaheba [6]: Temporarily takes ownership of a value using `Option::take()` instead of modifying it in place. With view types, in-place modification would be possible.

```
let mut terminator = mir
    .basic_block_data_mut(bb)
    .terminator
    .take()  // temporarily take ownership
    ...
```

4 Applications in Formal Verification

View types could also enhance formal verification in Rust. Verus [4], a verification framework for Rust, infers possible mutations from function signatures. A shared reference (`&T`) guarantees immutability[1], while a unique reference (`&mut T`) allows mutation of its value. Updating certain fields requires stating that others remain unchanged in the postcondition:

```
fn f(x: &mut (i32, i32)) ensures old(x.1) == x.1
    { x.0 = 5 }
```

With view types, this postcondition can move into the type, making the overall function signature more concise:

```
fn f(x: &{mut 0} (i32, i32)) { x.0 = 5 }
```

5 Soundness of View Types

Establishing the soundness of view types is crucial for its adoption in Rust. To this end, we aim to model view types within a formalization of Rust and prove key properties, most importantly *type soundness*. Type soundness ensures that a well-typed Rust program (i.e. one that passes type and borrow checking) will execute safely. While formalizing Rust remains an active research area, several models exist [1, 3, 8]. Among these, Aeneas [3] is particularly promising, as it provides a reusable transpiler from Rust, and it is actively maintained. We plan to use Aeneas as the foundation for our formal study of view types. Notably, Aeneas defines borrow checking via symbolic execution rather than traditional inference rules, and it will be interesting to see how easily it can be extended with view types.

6 Open Questions

Potential impact of view types. While we have identified over 15 real-world cases involving partial borrows, the broader impact of introducing view types remains uncertain, as community opinions are mixed [9]. To rigorously assess this, we aim to uncover hundreds or thousands of instances where view types could resolve borrow checking limitations. Achieving this at scale is challenging. One promising approach is to mine code comments for borrow checker complaints and derive code patterns that correlate with specific limitations. If some patterns connect to partial borrows, can be addressed by view types, and occur frequently in Rust functions, this would provide strong evidence for the practical impact of view types.

Abstraction barriers. Our current view types design is limited to private functions to avoid problems with encapsulation and semantic versioning. This restriction means that view types cannot be directly applied to widely used libraries like `std::vec::Vec`, whose public methods operate on private fields. Addressing this limitation, without compromising abstraction barriers, may require a clever design.

Soundness in practice. Proving the soundness of view types in a simplified Rust model does not guarantee their safety in real-world Rust, due to complexity and evolving features. Bridging this gap remains an important challenge.

7 Related Work

Macro-based libraries [2] provide DSLs that enable partial borrowing. They achieve this by rewriting code at compile time and inserting unsafe code, rather than extending Rust's type system. As a result, they lack first-class support, are thus less ergonomic, and do not provide formal soundness guarantees. This motivates the proposal for a language extension that integrates partial borrowing directly into Rust's type system.

References

[1] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 265 (Oct. 2023), 29 pages. doi:10.1145/3622841

[2] Wojciech Danilo. 2024. *Partial Borrows for Rust.* https://github.com/wdanilo/borrow

[3] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647

[4] 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. doi:10.1145/3586037

[5] Nicholas Matsakis. 2021. *View types for Rust.* https://smallcultfollowing.com/babysteps/blog/2021/11/05/view-types/

[6] Sasha Pak. 2025. *Partial Borrows Examples.* https://docs.google.com/spreadsheets/d/e/2PACX-1vSnJKtJ0wn9mJQ3GjQwr2S8hC4YXEpYTEGoU6L_4u7MZFnxiKsidjcdtdF63u46I5PxuwXbAmV0w_8G/pubhtml?gid=0&single=true

[7] Various. 2025. GitHub Borrow Checker Complaints. https://github.com/search?q=%22borrow+checker+happy%22&type=code. Accessed: 2025-08-19.

[8] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. arXiv:1903.00982 [cs.PL] https://arxiv.org/abs/1903.00982

[9] xmBQWugdxjaA. 2025. Thoughts on proposed View types by @nikomatsakis? https://www.reddit.com/r/rust/comments/1ja9ee8/thoughts_on_proposed_view_types_by_nikomatsakis/. Accessed: 2025-08-19.

[1] Verus does not verify interior mutability or direct unsafe code.

Received 2025-08-19; accepted 2025-08-24; revised 19 August 2025