Formal Methods in System Design
https://doi.org/10.1007/s10703-024-00460-3

# Formally understanding Rust's ownership and borrowing system at the memory level

**Shuanglong Kan[1] · Zhe Chen[2] · David Sanán[3] · Yang Liu[4]**

Received: 2 January 2023 / Accepted: 20 May 2024


**Abstract**
Rust is an emergent systems programming language highlighting memory safety through its Ownership and Borrowing System (OBS). Formalizing OBS in semantics is essential in certifying Rust's memory safety guarantees. Existing formalizations of OBS are at the language level. That is, they explain OBS on Rust's constructs. This paper proposes a different view of OBS at the memory level, independent of Rust's constructs. The basic idea of our formalization is mapping the OBS invariants maintained by Rust's type system to memory layouts and checking the invariants for memory operations. Our memory-level formalization of OBS helps people better understand the relationship between OBS and memory safety by narrowing the gap between OBS and memory operations. Moreover, it enables potential reuse of Rust's OBS in other programming languages since memory operations are standard features and our formalization is not bound to Rust's constructs. Based on the memory model, we have developed an executable operational semantics for Rust, called RustSEM, and implemented the semantics in K-Framework ($\mathbb{K}$). RustSEM covers a much larger subset of the significant language constructs than existing formal semantics for Rust. More importantly, RustSEM can run and verify real Rust programs by exploiting $\mathbb{K}$'s execution and verification engines. We have evaluated the semantic correctness of RustSEM wrt. the Rust compiler using around 700 tests. In particular, we have compared our formalization of OBS in the memory model with Rust's type system and identified their differences due to the conservation of the Rust compiler. Moreover, our formalization of OBS is helpful to identifying undefined behavior of Rust programs with mixed safe and unsafe operations. We have also evaluated the potential applications of RustSEM in automated runtime and formal verification for

✉ Zhe Chen
zhechen@nuaa.edu.cn

Shuanglong Kan
shuanglong@cs.uni-kl.de

David Sanán
david.miguel@singaporetech.edu.sg

Yang Liu
yangliu@ntu.edu.sg

1 Department of Computer Science, Technische Universität Kaiserslautern, Kaiserslautern, Germany

2 College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

3 InfoComm Technology Cluster, Singapore Institute of Technology, Singapore, Singapore

4 School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

Published online: 09 July 2024

functional and memory properties. Experimental results show that RustSEM can enhance Rust's memory safety mechanism, as it is more powerful than OBS in the Rust compiler for detecting memory errors.

**Keywords** Rust · Language Semantics · K-Framework · Formal Verification

## 1 Introduction

Developing formal semantics for a programming language could provide a mathematical foundation for the language. The semantics can be used as a reference model and, more importantly, as a foundation for proving language-level properties and constructing automated verification tools.

Rust [1] is an emergent systems programming language aiming at providing memory safety guarantees with its Ownership and Borrowing System (OBS). One of the most important guarantees the OBS invariants maintain is the exclusive mutation capability for memory locations, which can avoid various memory errors such as dangling pointers, double frees, and data races.

Several formal semantics for Rust has been developed. RustBelt [2] formalizes a variant $\lambda_{Rust}$ of Rust in Coq, to prove in Coq that the type system of Rust can guarantee the memory and thread safety of $\lambda_{Rust}$ programs. Patina [3] formalizes the semantics of an early version of Rust. Stacked Borrows [4] give an explanation of Rust's OBS using alias stacks and further define undefined behaviors of Rust.

All these existing works understand Rust's OBS at the *language-level*. In other words, their explanation of OBS is *bound to Rust's constructs*. For instance, RustBelt formalizes OBS using a type system over Rust's constructs. Even though the language-level OBS can provide clear instructions for users on how to follow OBS in Rust programming, we still need a deep insight into OBS over low-level memory management. Because language level can only present the effect of OBS over Rust's constructs, such as bindings and assignments, but memory-level OBS can directly illustrate the effect of OBS over memory operations, such as reading or writing a memory location.

In this paper, we propose a new executable operational semantics for Rust. The core of our semantics is a formal *memory model* that includes memory layouts and the semantics of memory operations. More importantly, the memory model gives a *memory-level* formalization of OBS, which is *independent of Rust's constructs*. Its basic idea is mapping the OBS invariants to a language-independent memory model. The OBS invariants are checked during the execution of memory operations. Besides the OBS invariants, the memory model formalizes the interaction between safe and unsafe operations in Rust. To show the correctness of the memory model, we define a high-level abstraction of OBS (HOBS) and have proved the refined relation between HOBS and the memory model.

The interests of the memory model are twofold. On the one hand, our memory-level formalization of OBS helps people understand the relationship between OBS and memory safety. On the other hand, our language-independent formalization of OBS makes it possible to reuse OBS in different programming languages, as memory manipulation is a common and essential feature of mainstream languages.

Based on the memory model, we have developed an executable operational semantics called RustSEM. RustSEM consists of an operational semantics for the Core Language (CL) and a semantics for the translation from Rust to CL. The CL is an intermediate representation

(IR) for avoiding redundant semantics definitions of Rust's constructs. Its semantics contain the semantics of memory operations. To execute a Rust program, RustSEM first translates the program into a corresponding CL program and then executes the CL program wrt. the CL semantics, in which all memory accesses are carried out by invoking the interfaces of the memory model. Recall that Rust also has an IR, namely Mid-level Intermediate Representation (MIR) [5]. However, the gap between Rust and CL is much smaller than MIR, simplifying the translation semantics from Rust to CL.

We have implemented RustSEM in the executable semantics modeling tool K-Framework (𝕂) [6]. 𝕂 is based on rewriting-logic. Thanks to its built-in parser, language semantics can be defined on abstract syntax trees. 𝕂 has been successfully applied in formalizing the semantics of real-world programming languages, such as Java [7] and C [8, 9]. 𝕂's execution and verification engines facilitate the testing of RustSEM and constructing automated verifiers.

RustSEM distinguishes from the existing semantics of Rust in the following aspects.

Firstly, RustSEM directly formalizes the semantics based on Rust's grammar instead of a variant, thus covering a much larger subset of the significant language constructs than existing semantics. For instance, RustSEM supports safe and unsafe constructs, concurrency, dynamic OBS, closures, pattern matching, and polymorphism.

Secondly, RustSEM is an executable semantics, which means that RustSEM can execute a Rust program concerning its semantics. This makes semantics-based runtime verification and formal verification possible.

Thirdly, thanks to the integrated memory model, RustSEM can verify the runtime behavior of Rust programs against the OBS invariants, i.e., reject the programs violating the OBS invariants. RustSEM can also detect undefined behavior (cf. Stacked Borrows [4]) of the programs with mixed safe and unsafe operations, by noting that unsafe constructs can escape from OBS and make programs error-prone [10],

We have evaluated the semantic correctness of RustSEM wrt. the Rust compiler, including semantic consistency (i.e., absence of ambiguities), functional correctness, and OBS implementation correctness, using around 700 tests which mainly come from the Rust benchmarks [11], the Rust libraries, and the Rust textbook [12]. In particular, we have proposed a new testing technique for detecting ambiguities, which has discovered more than 36 ambiguities in the early versions of RustSEM.

We have also evaluated the potential applications of RustSEM in automated runtime verification and formal verification for both functional and memory properties. The application in runtime verification is evaluated on 118 Rust programs for detecting memory errors. The application in formal verification is evaluated on a collection of benchmarks, including `Vec_Deque` in the Rust library implementing a ring buffer. Experimental results show that RustSEM can enhance Rust's memory safety mechanism, as it is more powerful than OBS in detecting memory errors.

In summary, we make the following contributions:

1. We propose a high-level abstraction of OBS and a memory model containing the operational semantics of OBS. We have also proved the refinement relation between them.
2. We propose a new executable operational semantics for Rust, based on the memory model. Our semantics supports a *larger* subset of the significant language constructs (compared with existing semantics). It contains the semantics of unsafe raw pointers and the executions with mixed safe and unsafe pointers.
3. We have evaluated the correctness of our semantics. In the evaluation, we have also proposed a novel testing technique based on 𝕂's verification engine for detecting ambiguities.

4. We show that RustSEM can be applied to both runtime and formal verification against functional and memory properties.

This paper is organized as follows: Sect. 2 recalls the OBS of Rust. Section 3 presents a high-level abstraction of OBS. Section 4 defines the operational semantics of the memory model while Sect. 5 presents the basic idea of the semantics of CL and the translation semantics from Rust to CL. Section 6 evaluates the proposed semantics. Section 7 compares related work. Section 8 concludes.

## 2 The ownership and borrowing system

In this section, we recall Rust's Ownership and Borrowing System (OBS) and the related OBS invariants.

### 2.1 Ownership

A variable can declare a memory block's unique *ownership* using a binding or an assignment. The owner can read and write the block if the ownership is declared mutable with the `mut` keyword. Otherwise, it is read-only. A read-only (resp. mutable) owner is called a *shared* (resp. *mutable*) *alias* of the block. We denote by $\texttt{x} \rightarrow_o \texttt{B}$ that variable `x` is the owner of block `B`, i.e., `x` owns `B`. For instance, in Listing 1, the binding "`let mut v = vec![1,2]`" at Line 1.1 first allocates a block `B` in the memory to store the vector `[1,2]`, and then the owner `v` obtains the ownership of `B`, denoted by $\texttt{v} \rightarrow_o \texttt{B}$.

**Listing 1** Ownership

```
1.1 let mut v = vec![1,2];        } v
1.2 {
1.3     let v1 = v;               }
1.4     let t = v1[0];            } v1
1.5     v1[1] = 3;                }
1.6 }
```

An ownership can be *moved* from one variable to another. Moving an ownership from variable $x$ to another variable $y$ means that the ownership now belongs to $y$ and $x$ no longer owns it. For instance, in Listing 1, the binding "`let v1 = v`" at Line 1.3 moves the ownership of the vector from `v` to `v1`, i.e., `v1` becomes the new owner of the vector and `v` can no longer be used to access it. Indeed, the vector is read and written through `v1` at Lines 1.4 and 1.5, respectively.

The *timestamp* is a way to distinguish the execution order of program statements. This section uses line numbers as timestamps. The *lifetime* of an owner begins from the timestamp at which it obtains the ownership and ends at the timestamp at which it loses the ownership, e.g., when the ownership is moved, or it goes out of the program scope (i.e., curly braces). The owned block is automatically deallocated through its owner when the owner goes out of scope. For instance, in Listing 1, the lifetime of owner `v` begins at Line 1.1 and ends at Line 1.3, whilst the lifetime of owner `v1` begins at Line 1.3 and ends at Line 1.6. The vector is deallocated through `v1` at Line 1.6 as `v1` goes out of scope.

## 2.2 Borrowing and reborrowing

*Borrowing* is a way to create *references* from the owner of a memory block. There are two kinds of references: *shared* references (read-only, created by `&`) and *mutable* references (readable and writable, created by `&mut`). A shared (resp. mutable) reference is also called a *shared* (resp. *mutable*) *alias* of the block.

A borrowing creating a shared (resp. mutable) reference is called a shared (resp. mutable) borrowing. We denote by $x \rightarrow_s y$ (resp. $x \rightarrow_m y$) that $x$ is a shared (resp. mutable) reference to $y$, i.e., $x$ borrows $y$. For instance, in Listing 2, the borrowing "`let b1 = &v[0]`" at Line 2.3 creates a shared reference `b1` to `v`, denoted by `b1` $\rightarrow_s$ `v`, where `b1` can only be used to read the memory location `v[0]` owned by `v`. The borrowing "`let b2 = &mut v`" at Line 2.4 creates a mutable reference, denoted by `b2` $\rightarrow_m$ `v`, where `b2` can be used to both read and write the block owned by `v`.

**Listing 2** Borrowing

```
2.1 let mut v = vec![1,2];
2.2 ......
2.3 let b1=&v[0];
2.4 let b2=&mut v;
2.5 (*b2).push(2);
2.6 let t = *b1;
```

`b2` $\rightarrow_m$ `v` (Lines 2.4–2.5); `b1` $\rightarrow_s$ `v` (Lines 2.3–2.6)

*Reborrowing* is a way to create *references* from another reference, instead of an owner. For instance, in Listing 3, `b1` borrows the owner `v` at Line 3.2 and `b2` reborrows `b1` with the referent `*b1` at Line 3.3 , denoted by the link `b2` $\rightarrow_s$ `b1` $\rightarrow_m$ `v`. To access the first element of the vector, we can use `v[0]`, `(*b1)[0]` or `(*b2)[0]`, where `v`, `*b1` and `*b2` are three paths to access the vector. Each path uses an alias as the entry to access the block. For instance, the paths `v`, `*b1` and `*b2` use the alias entries `v`, `b1` and `b2`, respectively.

**Listing 3** Reborrowing

```
3.1 let mut v = vec![1,2];
3.2 let b1=&mut v;
3.3 let b2=&(*b1);
3.4 let z=(*b2)[0];
3.5 (*b1).push(2);
3.6 let t = (*b2)[1];
```

`b1` $\rightarrow_m$ `v` (Lines 3.2–3.5); `b2` $\rightarrow_s$ `b1` (Lines 3.3–3.6)

As borrowing and reborrowing are similar, we define unified relations for them.

**Definition 1** (*Unified borrowing relations*) Let $x$ and $y$ be two variables, where $y$ can be either an owner or a reference. $x$ borrows or reborrows $y$ is denoted by $x \rightarrow_b y$. The borrowing relation $\rightarrow_b$ contains two sub-relations: *shared* borrowing $\rightarrow_s$ and *mutable* borrowing $\rightarrow_m$, i.e., $x \rightarrow_b y \iff (x \rightarrow_s y) \vee (x \rightarrow_m y)$.

The *lifetime* of a reference begins from the timestamp of its creation and ends at the last timestamp at which it is used (read, written, or borrowed), according to the definition of

Non-Lexical Lifetimes (NLL) [13]. Unlike the owner's lifetime, which the program scope can easily decide, the last use of it decides the lifetime of a reference. For instance, in Listing 2, the lifetime of `b1` is from Line 2.3 to Line 2.6 , while the lifetime of `b2` is from Line 2.4 to Line 2.5 .

### 2.3 The OBS invariants

Because multiple aliases can access a memory block, the OBS should ensure that a Rust program fulfills the following guarantees

1. Each alias only accesses (reads or writes) one valid block. For owners, they should own the block. For references, their owners should be in their lifetimes to ensure validity.
2. At any time in execution, each block can be accessed by either multiple shared aliases (but no mutable alias) or exclusively accessed by a unique mutable alias.

These guarantees can be used to avoid memory errors such as dangling pointers (guarantee 1) and data races (guarantee 2).

To meet these two guarantees, the OBS checks Rust programs against a collection of invariants. We have summarized the following five invariants by studying the Rust documents and the compiler implementation. We will prove that these invariants precisely ensure the guarantees in Sect. 3.

**Definition 2** (*The OBS invariants*) The following invariants must be satisfied:

1. *Unique owner invariant*. Each block has a unique owner. The block is deallocated only when the owner's lifetime ends.
2. *Lifetime inclusion invariant*. If $x \rightarrow_b y$, then the lifetime of $x$ should always be within the lifetime of $y$.
3. *Lifetime disjoint invariant*. There are *no* two direct references to the same referent such that their lifetimes intersect, and one of them is a mutable reference. For example, if $x \rightarrow_m y$ and $z \rightarrow_b y$, then the lifetimes of $x$ and $z$ should not intersect.
4. *Write permission disabled invariant*. If $x \rightarrow_s y$, then the write permission of $y$ should be turned off until the end of $x$'s lifetime.
5. *Full permission disabled invariant*. If $x \rightarrow_m y$, then both the read and write permissions of $y$ should be turned off until the end of $x$'s lifetime.

We now illustrate the above invariants by examples. The unique owner invariant is obvious. The move operation preserves the invariant.

**Listing 4** Invariant (2) Violation

```
4.1 let mut b;
4.2 {
4.3     let mut v = vec![1,2];
4.4     b = &mut v;
4.5     let t=(*b)[1];
4.6 }
4.7 (*b)[1] = 2;
```

b →m v

v

Listing 4 violates Invariant (2). The lifetime of owner `v` is from Line 4.3 to 4.6 and the lifetime of reference `b` is from Line 4.4 to 4.7 . Therefore, the reference's lifetime is not within the owner's, leading to a dangling pointer.

Listing 2 violates Invariant (3). The lifetime of `b1` $\rightarrow_s$ `v` is from Line 2.3 to 2.6 . The lifetime of `b2` $\rightarrow_m$ `v` is from Line 2.4 to 2.5 . The intersection of the lifetimes of `b1` $\rightarrow_s$ `v` and `b2` $\rightarrow_m$ `v` is not empty and one of them is a mutable reference. This is a potential vulnerability. At Line 2.5 , if the memory space of the vector is already full then the push operation will reallocate a memory block for the vector. But the reference `b1` still points to the old memory block and is used to read at Line 2.6 , resulting in a dangling pointer.

Listing 3 violates Invariant (4). The lifetime of shared reference `b2` is from Line 3.3 to 3.6 . Since `b2` reborrows `b1`, `b1`'s write permission should be disabled within the lifetime of `b2`. Thus, the write through `b1` at Line 3.5 is illegal.

### 2.4 Unsafe constructs

Note that the OBS only concerns *safe pointers* such as owners and references. Recall that Rust also supports *unsafe raw pointers*, which can escape from OBS checking. Unsafe raw pointers are widely used in real-world Rust programs to facilitate programming but may introduce more memory errors. Raw pointers do not need to preserve the OBS invariants, but they need to know safe pointers' ownership and borrowing information. For instance, when a raw pointer tries to read through an owner, we need to know whether the ownership has been moved away to check dangling pointers. We will formalize raw pointers in our memory model in Sect. 4.

Besides unsafe pointers, Rust also supports other unsafe features, including *unsafe scopes* and *unsafe functions*, in which unsafe raw pointers are used. We will model these unsafe features in our Core Language and translation semantics in Sect. 5.

## 3 A high-level abstraction of OBS

In this section, we propose a new high-level abstraction of OBS, which formalizes the ownership and borrowing relations among aliases and memory blocks as graphs. At the end of this section, by using this abstraction, we will formalize the OBS invariants in Sect. 2.3 and prove that they precisely ensure the OBS guarantees.

**Definition 3** (*Timestamp*) Let $Tim$ be an infinitely countable set of timestamps. $(Tim, \leq)$ is a Totally Ordered Set (TOS). A strict order $<$ over $Tim$ is defined as $t < t'$ iff $t \leq t' \wedge t \neq t'$. The function $Su : Tim \rightarrow Tim$ is the successor function of timestamps, such that $Su(t) = t'$ iff $t < t' \wedge \nexists t''.t < t'' < t'$.

**Definition 4** (*Lifetime*) A lifetime $t_1 \sim t_2$, where $t_1, t_2 \in Tim$, is defined as $\{t \mid t_1 \leq t \leq t_2\}$, which is a subset of $Tim$. Let $LT$ be the set of all lifetimes.

An OBS graph consists of a memory block, its aliases and their relations.

**Definition 5** (*OBS graph*) An OBS graph $G$ is a tuple $(B, V, E, \mathcal{F})$, where

- $B$ is a memory block.
- $V$ is a set of aliases of $B$, e.g., owners and references.

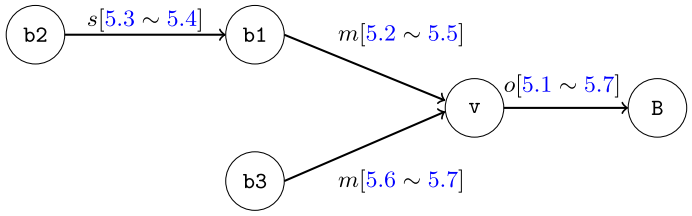



**Fig. 1** The OBS graph of the program in Listing 5

- $E : V \to \{o, s, m\} \times (\{B\} \cup V)$ is a set of edges, i.e., a total mapping from aliases to the product of $\{o, m, s\}$ and $\{B\} \cup V$, where $E(a) = (o, B)$ iff $a \to_o B$, i.e. $a$ owns $B$; $E(a) = (s, a')$ iff $a \to_s a'$, i.e., $a$ is a shared reference to $a'$; $E(a) = (m, a')$ iff $a \to_m a'$, i.e., $a$ is a mutable reference to $a'$.
- $\mathcal{F} : E \to LT$ is the lifetimes of edges.

**Listing 5** An example demonstrating OBS graphs

```
5.1 let mut v = vec![1,2,...,n];
5.2 let b1 = &mut v;
5.3 let b2 = &(*b1);          } b2 →s b1   } b1 →m v
5.4 println!((*b2)[0]);
5.5 (*b1)[0] = 2;
5.6 let b3 = &mut v;          } b3 →m v
5.7 (*b3)[0] = 3;
```

Note that $E$ is a total mapping, as each alias holds exactly one value. In sequel, $E(a) = (*, a')$ and $(a \to_* a') \in E$ are used interchangeably, where $* \in \{o, s, m\}$. We also reuse the notation $\to_b$ in Sect. 2 to denote either $\to_m$ or $\to_s$.

Figure 1 shows an OBS graph corresponding to the program in Listing 5, where `B` denotes the memory block that stores the vector `vec![1,2,...,n]`. Each edge is labeled with the mutability of the borrowing and its lifetime. For instance, the edge from `b2` to `b1` is labeled with $s[5.3 \sim 5.4]$, which denotes `b2` $\to_s$ `b1` and $\mathcal{F}$(`b2` $\to_s$ `b1`) = $5.3 \sim 5.4$.

To formalize the OBS invariants in Definition 2, we define well-formed OBS graphs (for Inv 1, 2, 3) and the read and write permission functions (for Inv 4, 5).

**Definition 6** (*Well-formed OBS graph*) An OBS graph $G = (B, V, E, \mathcal{F})$ is well-formed if and only if it satisfies:

1. There is a unique $a \in V$ such that $E(a) = (o, B)$.
2. $\forall a, a', a'' \in \{B\} \cup V.\ E(a) = (*, a') \wedge E(a') = (*, a'') \implies \mathcal{F}(a \to_* a') \subset \mathcal{F}(a' \to_* a'')$.
3. $\forall a, a', a'' \in V.\ E(a') = (b, a) \wedge E(a'') = (m, a) \implies \mathcal{F}(a' \to_b a) \cap \mathcal{F}(a'' \to_m a) = \emptyset$.

For example, the OBS graph in Fig. 1 is well-formed. Indeed, the block `B` is uniquely owned by `v`. The lifetime of `b2` $\to_s$ `b1` is within the lifetime of `b1` $\to_m$ `v`. The lifetimes of `b1` $\to_m$ `v` and `b3` $\to_m$ `v` do not intersect. One property of a well-formed OBS graph is that it is acyclic. The proof is in Appendix A.

**Lemma 1** *A well-formed OBS graph is acyclic.*

The read and write permission functions permit an alias to read or write at a specific timestamp only if the read or write permission function returns true, respectively.

**Definition 7** (*Read and write permission functions*) Let $G = (B, V, E, \mathcal{F})$ be an OBS graph. The read permission function $R_G : V \times Tim \to \mathbb{B}$ is defined as

$$R_G(a,t) = \begin{cases} true & if\ (\exists a'.a \to_* a' \in E \wedge t \in \mathcal{F}(a \to_* a')) \wedge (\nexists a''.t \in \mathcal{F}(a'' \to_m a)) \\ false & otherwise \end{cases}$$

where $t \in \mathcal{F}(a \to_* a')$ ensures that $t$ is in the lifetime of $a$ and $\nexists a''.t \in \mathcal{F}(a'' \to_m a)$ ensures that $a$'s read permission is not disabled at $t$.

The write permission function $W_G : V \times Tim \to \mathbb{B}$ is defined as

$$W_G(a,t) = \begin{cases} true & if\ (\exists a'.a \to_{\{o,m\}} a' \in E \wedge t \in \mathcal{F}(a \to_{\{o,m\}} a')) \wedge (\nexists a''.t \in \mathcal{F}(a'' \to_b a)) \\ false & otherwise \end{cases}$$

where $a \to_{\{o,m\}} a'$ denotes $a \to_o a'$ or $a \to_m a'$.

For example, in Fig. 1, $W_G(\texttt{b1}, 5.3) = false$ as there exists b2 such that $5.3 \in \mathcal{F}(\texttt{b2} \to_s \texttt{b1})$. $R_G(\texttt{b3}, 5.7) = true$ as $5.7 \in \mathcal{F}(\texttt{b3} \to_m \texttt{v})$ and $\nexists a'.a' \to_b \texttt{b3} \in E$.

The relations between the permission functions in Definition 7 and Inv 4 and 5 in Definition 2 can be specified as: (1) an alias's reading (resp. writing) operation is disabled at the timestamp $t$ due to Inv 4 or 5 if and only if the read (resp. write) permission function for the alias at $t$ returns false.

We can show in the following two theorems that a well-formed OBS graph satisfying the permission functions precisely ensures the OBS guarantees. The proof is in Appendix B.

**Theorem 1** *Let $G = (B, V, E, \mathcal{F})$ be a well-formed OBS graph satisfying the permission functions and $t$ be a timestamp. We have either (1) $\forall a \in V, W_G(a,t) = false$ or (2) $\exists! a \in V, W_G(a,t) = true \wedge \forall a'.(a \neq a' \Rightarrow R_G(a',t) = W_G(a',t) = false)$. The notation $\exists!$ denotes unique existential quantification.*

**Theorem 2** *Let $G = (B, V, E, \mathcal{F})$ be a well-formed OBS graph satisfying the permission functions and $t$ be a timestamp. For any $a \in V$, if $R_G(a,t) = true$ or $W_G(a,t) = true$, then $a$ is not a dangling pointer.*

In other words, Theorem 1 shows that the exclusive mutation guarantee is fulfilled if all reads and writes performed by an alias are executed at the timestamps when the corresponding permission functions return true. Theorem 2 shows that the memory block accessed by an alias is valid at the timestamp when the permission function returns true.

**Discussions.** Our high-level abstraction of OBS is language-independent and flexible.

Firstly, the definitions of OBS graphs and permission functions do not use Rust's constructs. This is potentially helpful to reuse OBS in other programming languages, such as C, to improve their memory safety.

Secondly, there is another definition of lifetime in Rust, called two-phase borrowing [14], in which the lifetime of a mutable reference starts from the timestamp of its first use instead of the timestamp of its creation. Our high-level abstraction is also compatible with this definition by simply overriding the lifetime definition of reference, For instance, in Fig. 1, with two-phase borrowing, $\mathcal{F}(\texttt{b3} \to_m \texttt{v}) = 5.7 \sim 5.7$. Definitions 6 and 7 remain unchanged. Theorems 1 and 2 still hold.

## 4 The memory model

In this section, we introduce our memory model, the core of RustSEM that formalizes OBS. It can be viewed as an implementation or a refinement of the high-level abstraction of OBS. The memory model supports the dynamic checking of OBS invariants. Moreover, it also supports sequential consistency checking for concurrent accesses.

We first illustrate *dynamic lifetime extension* using a motivating example in Fig. 2. Line 6.1 creates a block $B$ for `vec![1,2]` and assume the block location of $B$ is `b`. An ownership relation `v` $\rightarrow_o$ $B$ that variable `v` is the owner of $B$ is also created. The notation `v` $\mapsto$ `own(b)` denotes that variable `v` starts to own the block whose location is `b` where `own(b)` indicates the ownership of $B$. Line 6.2 creates a borrowing relation `b1` $\rightarrow_m$ `v`. We introduce value `mut`$(6.2 \sim 6.2,$ `v`$)$ to denote a *reference value* to `v` in the memory model, where $6.2 \sim 6.2$ is a timestamp span in which `b1` is used, since we have only scanned the code up to Line 6.2. When we scan Line 6.3, `b1` is used to write the vector, thus the timestamp span of the reference `b1` should be extended. The new span is $6.2 \sim 6.3$. This treatment of lifetimes is called *dynamic lifetime extension*. Line 6.4 writes the vector via its owner `v`. Note that `b1` does not disable `v` here since its lifetime has not been extended to Line 6.4. Line 6.5 creates a relation `b2` $\rightarrow_s$ `v`. We introduce the value `shr`$(6.5 \sim 6.5,$ `v`$)$ to denote a shared reference value to `v`, whose lifetime is from Line 6.5 to 6.5 at this moment. Line 6.6 writes the vector via its owner; this is allowed at this moment since the existing two references cannot disable it up to now. But at Line 6.7, since `b2` is used to read, the lifetime of the reference `b2` should be extended to 6.7. Now, an error occurs, i.e., the write by `v` at Line 6.6 is within the lifetime of the shared reference `b2`.

This motivating example shows that we need to store the lifetimes of references to support dynamic OBS invariant checking without looking backward. Moreover, we also need to record the timestamps at which an alias is used to read or write. For instance, in Fig. 2 the owner `v` is recorded to be used to write at Lines 6.4 and 6.6.

Figure 3 illustrates the grammar of the memory model. Note that semantic rules generate timestamps instead of line numbers. We will explain these constructs in the following subsections.

Listing 6: Motivation for Dynamic Lifetime Extension

| | | |
|---|---|---|
| **6.1** | `let mut v = vec![1,2];` | `v` $\mapsto$ `own(b)` |
| **6.2** | `let b1 = &mut v;` | `b1` $\mapsto$ `mut(6.2`$\sim$`6.2, v)` |
| **6.3** | `(*b1)[0] = 2;` | `b1` $\mapsto$ `mut(6.2`$\sim$`6.3, v)`, `b1` is used to write |
| **6.4** | `v[0]=1;` | `v` is used to write |
| **6.5** | `let b2 = & v;` | `b2` $\mapsto$ `shr(6.5`$\sim$`6.5,v)` |
| **6.6** | `v[1] = 2;` | `v` is used to write |
| **6.7** | `let t = (*b2)[1]` | `b2` $\mapsto$ `shr(6.5`$\sim$`6.7,v)` |

**Fig. 2** A motivating example for dynamic lifetime extension. An error exists at Line 6.7, since `b2` should disable the write permission of `v` during the lifetime $6.5 \sim 6.7$, but `v` writes at Line 6.6

| | | |
|---|---|---|
| Natural number | $\mathtt{n}, \mathtt{m} \in \mathbb{N}$ | |
| Types | $\mathtt{t} \in \mathcal{T}$ | |
| Time stamps | $\mathtt{ts} \in Tim$ | |
| Local Lifetime | $lt \in LT$ | $::= \mathtt{ts}_1 \sim \mathtt{ts}_2$ |
| Stack location | $\mathtt{s} \in L_s$ | $(L_s, \leq_s)$ is a POR |
| Block location | $\mathtt{b} \in L_b$ | $(L_b, \leq_b)$ is a POR |
| Heap location | $(\mathtt{b}, \mathtt{n}) \in L_h$ | $L_h \triangleq L_b \times \mathbb{N}$ |
| Mem location | $l \in L_m$ | $::= \mathtt{s} \mid (\mathtt{b}, \mathtt{n})$ |
| Paths | $p$ | $::= \mathtt{s} \mid *p \mid p.\mathtt{n}$ |
| Pointer Value | $pv$ | $::= \mathtt{own}(\mathtt{b}) \mid \mathtt{raw}(l) \mid \mathtt{shr}(lt, p)$ |
| | | $\mid \mathtt{shr}(p) \mid \mathtt{mut}(lt, p) \mid \mathtt{mut}(p)$ |
| Primitive Value | $sv \in SV$ | $::= i \in \mathtt{Int} \mid bv \in \mathtt{bool} \mid \ldots$ |
| Values | $v \in \mathtt{Val}$ | $::= \perp \mid pv \mid sv$ |

Mem Operation :

$mop ::= \mathtt{alloc(n,t)} \mid \mathtt{free(b)} \mid \mathtt{lv}(p) \mid \mathtt{read}(p) \mid \mathtt{write}(p, v)$
$\mid \mathtt{rawRead}(l) \mid \mathtt{rawWrite}(l, v) \mid \mathtt{aRead}(l) \mid \mathtt{aWrite}(l, v)$

**Fig. 3** The grammar of the memory model

**Listing 7** An example of memory layout

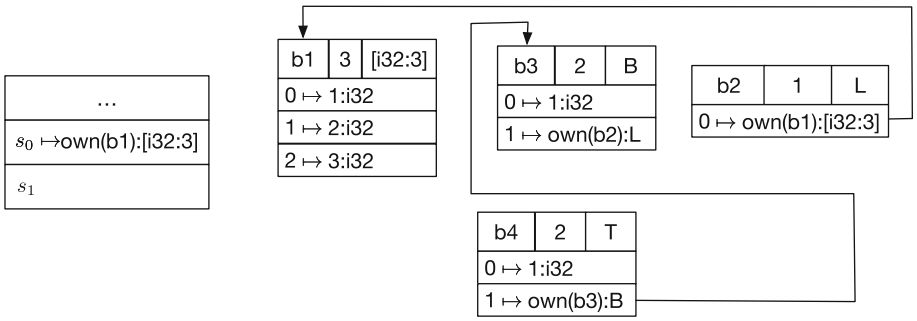
```
7.1 struct T {x:i32,y:B}
7.2 enum B {Empty,L([i32:3])}
7.3 let a = [1,2,3];
7.4 let b = B::L(a);
7.5 let t = T{x:1,y:b};
```

### 4.1 Memory configurations

A memory configuration is defined as a 4-tuple $mem = (S, H, \mathcal{P}, ms)$, where $S$ is a set of stacks, $H$ is the heap, $\mathcal{P}$ stores the latest timestamps at which alias is used to read or write as we explained in the motivating example, $ms$ is used for sequential consistency checking of concurrent access. We elaborate on them as follows.

The stacks are modeled as a finite partial map $S : L_s \xrightarrow{fin} (\mathtt{Val} \times \mathcal{T})$ from stack locations in the set $L_s$ ("Stack locations" in Fig. 3. POR denotes Partially Ordered Set) to typed values, denoted as $v : \mathtt{t}$, where $v \in \mathtt{Val}$ is a value and $\mathtt{t} \in \mathcal{T}$ is its type. Different stacks have disjoint location spaces in $L_s$ to ensure they are local to their corresponding threads. $L_s$ is partial order under $\leq_s$ since only stack locations in the same thread are ordered to denote the creation order of local variables. Various strategies could be selected to implement $L_s$. For instance, a stack location could be a pair $(tid, \mathtt{n})$, where $tid$ is a thread id and $\mathtt{n}$ is a natural number to denote a local stack location of $tid$.

The heap is defined as a set of blocks $H = \{B_1, \ldots, B_n\}$. A block $B_i$ is 4-tuple $(\mathtt{b}_i, \mathtt{n}_i, m_i, \mathtt{t}_i)$, where $\mathtt{b}_i \in L_b$, $L_b$ is a set of block locations ("Block locations" in




1. **struct** T {x:i32,y:B}
2. **enum** B {Empty,L([i32:3])}
3. **let** $a$ = [1,2,3];
4. **let** $b$ = B::L($a$);
5. **let** $t$ = T{x:1,y:$b$};

**Fig. 4** An example of the stacks and heap

Fig. 3) and each block has a unique block location, $\mathtt{n}_i \in \mathbb{N}$ is the size of the block, $m_i : [0, \mathtt{n}_i - 1] \to \mathtt{Val} \times \mathcal{T}$ is a map from the offsets in the range $[0, \mathtt{n}_i - 1]$ to the corresponding typed values ($[\mathtt{n}_1, \mathtt{n}_2]$ denotes the set of natural numbers $\mathtt{n}_1, \mathtt{n}_1 + 1, \ldots, \mathtt{n}_2$), and $\mathtt{t}_i \in \mathcal{T}$ is the block's type. A heap location is defined as a pair $(\mathtt{b}, \mathtt{m})$ ("Heap locations" in Fig. 3), where b is a block location and m is the offset within the block b.

The definitions of stacks and heaps are capable of storing values of various types, such as primitive types (integers, boolean values, among others), pointers, arrays, product types, and sum types. Heap blocks storing values of primitive types or pointers have size 1. Blocks storing arrays or values of product types have size greater or equal than 1, according to the number of elements the types are composed of, and two elements in the case of sum types: one for the value itself and another to indicate the constructor that the sum type selects to construct the value. Figure 4 illustrates the stacks and heap created by the program in Listing 7.

Three variables `a,b,t` created at lines 7.3 , 7.4 , and 7.5 have the corresponding stack locations $s_0, s_1, s_2$, respectively. Line 7.1 and 7.2 define a `struct` type and an `enum` type, respectively. Line 7.3 creates a block to store an array of the type `[i32:3]` (a 32-bits integer array of the size 3). The location, size, and type of the block are `b1`, `3`, and `[i32:3]` respectively. The offsets 0, 1, 2 store the values 1, 2, 3 of type `i32`, respectively. Line 7.4 creates two blocks `b2` and `b3`. The block `b2` stores the value of the type `L` defined by the constructor `L` in the enumeration type `B`. The symbol `own(b1)` denotes a value storing the address of the block `b1`. The block `b3` stores the value of the type `B`, which has two constructors: `Empty` and `L`. The value "1" at the offset 0 in the block `b3` indicates the constructor `L` is selected, otherwise if it is 0 then the constructor `Empty` is selected. Line 7.5 creates the block `b4` storing the value of type `T`.

In order to access memory locations, paths are introduced (Paths in Fig. 3). A path could be a stack location, a dereference, or a field. Consider a struct type: `"struct P{x:i32,y:i32}"` and the code `"let v = P{x:1,y:2}; let z = & v; "`, if we want to access the field `x` of `v` with the alias `z` then the path is $(*\mathtt{z}).\mathtt{x}$. Assume the stack location of `z` is `s` and the field `x` corresponds to the offset 0. The path in the memory model is `(* s)`.0.

The third element in the configuration $mem$ is a finite partial map $\mathcal{P} : L_m \xrightarrow{fin} \text{Tim} \times \text{Tim}$ from memory locations ("Memory locations" in Fig. 3) to pairs of timestamps, which stores the latest timestamps at which a location was used to read and write, respectively. For instance, $\mathtt{s} \mapsto (\mathtt{ts}_1, \mathtt{ts}_2) \in \mathcal{P}$ means that the latest read and write using $\mathtt{s}$ happened at $\mathtt{ts}_1$ and $\mathtt{ts}_2$, respectively. This information is used to check whether a read or write is disabled by other references (cf. the motivating example).

Finally, $ms : L_m \xrightarrow{fin} \mathbb{N} \times \mathbb{N}$ is a finite partial map from memory locations to pairs of natural numbers, used to detect data races with respect to sequential consistency, where $l \mapsto (\mathtt{n}_1, \mathtt{n}_2) \in ms$ represents that there are $\mathtt{n}_1$ reads and $\mathtt{n}_2$ writes that simultaneously access from/to the location $l$. This idea is inspired by Jung et al. [2].

At the end of this subsection, we introduce some notations for $mem = (S, H, \mathcal{P}, ms)$. For a partial map $M$, $M(k)$ denotes value of the key $k$ in the map. If $k$ has no value in $M$ then $M(k)$ is undefined. The notation $M[k \leftarrow v]$ denotes a new map obtained from $M$ by replacing the value of $k$ with $v$, which is defined as: $M[k \leftarrow v](k') \triangleq \begin{cases} v & k = k' \\ M[k'] & k \neq k' \end{cases}$. Let $B = (\mathtt{b}, \mathtt{n}, m, \mathtt{t})$ be a block in $H$ and $l = (\mathtt{b}, \mathtt{n}'), 0 \leq \mathtt{n}' < \mathtt{n}$. $H(l) \triangleq m(\mathtt{n}')$ denotes the value stored in the heap location $l$. $H[l \leftarrow v] \triangleq (H \backslash \{B\}) \cup \{(\mathtt{b}, \mathtt{n}, m[\mathtt{n}' \leftarrow v], \mathtt{t})\}$ denotes a new heap by replacing the value in $l$ with $v$. Moreover, we use $mem(l) = v$ to denote $S(l) = v \vee H(l) = v$ and $mem[l \leftarrow v]$ to denote a new memory configuration in which $S$ is replaced with $S[l \leftarrow v]$ or $H$ is replaced with $H[l \leftarrow v]$ when $l$ is a stack or a heap location, respectively.

### 4.2 Memory values

We now introduce the kinds of values used in the memory. Generally speaking, there are three kinds of values: primitive values, pointers, and $\perp$ denoting the uninitialization of a memory location. Primitive values include integers, floating-pointer values, boolean values, characters, and strings, which are standard. Here we focus on elaborating pointer values ("Pointer Values" in Fig. 3). Pointer values consist of the following 3 kinds:

1. Own pointers ($\mathtt{own}(\mathtt{b})$) to indicate the ownership of a block $\mathtt{b}$, a location holds an $\mathtt{own}$ pointer means it owns the block.
2. Shared and mutable reference values. The shared (resp. mutable) reference values have two forms $\mathtt{shr}(lt, p)$ and $\mathtt{shr}(p)$ (resp. $\mathtt{mut}(lt, p)$ and $\mathtt{mut}(p)$). The reason is that lifetimes are transparent for users using the model, whilst, in the memory, a reference should be attached with a lifetime for OBS checking.
3. Unsafe pointers ($\mathtt{raw}(l)$), which are raw pointers to memory locations. Unsafe pointers belong to the unsafe features of Rust.

Own pointers and shared and mutable reference values are safe pointers. Rust distinguishes safe and unsafe pointers to decide whether to carry out OBS invariant checking.

The values $\mathtt{shr}(p)$ and $\mathtt{mut}(p)$ are reference values to $p$. For instance, we can write a borrowing like "`let x = & (*v);` " in Rust. Assume the stack locations of `v` and `x` are `s` and `s1` respectively then the reference value to be assigned to `s1` (the location of `x`) is $\mathtt{shr}(*\mathtt{s})$. Moreover, we have the relation $\mathtt{s1} \rightarrow_s \mathtt{s}$ created by the binding statement.

For two locations $l$ and $l'$ in the memory $mem$, if $l$ is a shared (resp. mutable) reference to $l'$ then we write $l \rightarrow_s l' \in mem$ (resp. $l \rightarrow_m l' \in mem$). The notation $\mathcal{L}(l, mem)$ denotes the lifetime of $l \rightarrow_b l'$.

### 4.3 Lifetime-free memory operation interfaces

The memory operation interfaces ("Memory Operations" in Fig. 3) are lifetime-free, which means that the parameters of memory operations have no lifetimes. This design aims at abstracting lifetime information from the memory interfaces to enable reusability, since other languages may not have the notion of lifetimes. Memory operations include:

1. Allocation (`alloc(n, t)`), allocates a new memory block of size `n` for storing the value of the type `t`. Free (`free(b)`), deallocates the memory block `b`.
2. Raw read and write (`rawRead`($l$), `rawWrite`($l$, $v$)) provide non-atomic read and write without integrating OBS invariant checking.
3. Atomic read and write (`aRead`($l$), `aWrite`($l$, $v$)) provide atomic read and write without integrating OBS invariant checking.
4. Safe read and write (`read`($p$), `write`($p$, $v$)) provide read and write with the integration of OBS invariant checking.
5. Lvalue (`lv`($p$)), computes the Lvalue of the path $p$. Lvalue of a path is the memory location identified by the path that is to be read or written and Rvalue is the value stored in the Lvalue of the path.

Here we need to restrict that if the parameter `v` in either `rawWrite`, `aWrite`, or `write` is a reference value then it can only be `shr`($p$) or `mut`($p$), i.e., no lifetimes. Actually, lifetime computation is hidden in the implementation of memory operations, which will be introduced in the following.

### 4.4 Operational semantics for memory operations

The semantics is defined by two kinds of transition relations. The first one is $\langle mem, tm\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem', tm'\rangle^{\mathtt{ts}'}$. $tm$ and $tm'$ are terms that can be a value, a memory operation, or "." indicating that the operation is consumed (empty sequence). The symbols `ts` and `ts`$'$ are the timestamps of the pairs and satisfying $\mathtt{ts} \leq \mathtt{ts}'$. The second one is $\langle mem, tm\rangle^{\mathtt{ts}} \rightsquigarrow_m$ `stuck` indicating the semantics gets stuck. Some of $S$, $H$, $\mathcal{P}$, $ms$ can be omitted in rules if they are not used to make rules more concise.

Rule ALLOCATION defines the semantics for `alloc(n, t)`, where `n` and `t` are the size and type of the new block, respectively. It creates a new block with a fresh block location ($fresh(H)$) and adds the new block to the heap $H$. Moreover, it also initializes $ms$ for the new block. `initBlk(n)` denotes the map $\{0 \mapsto \bot, \ldots, \mathtt{n}-1 \mapsto \bot\}$. `initMS(b,n)` = {`(b,0)` $\mapsto$ `(0,0)`, ..., `(b,n-1)` $\mapsto$ `(0,0)`}. FREE removes a block from $H$. The set $bLoc(H)$ denotes all the block locations used in $H$, i.e., for any $\mathtt{b} \in bLoc(H)$, there is a block (b, n, $m$, t) in $H$. "_" matches anything. The timestamps are increased by $Su(\mathtt{ts})$.

$$\text{ALLOCATION:}\quad \frac{\mathtt{b} = fresh(H) \quad H' = H \cup (\mathtt{b}, \mathtt{n}, \mathtt{initBlk(n)}, \mathtt{t}) \quad ms' = ms \cup \mathtt{initMS(b,n)}}{\langle (H, ms), \mathtt{alloc(n,t)}\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle (H', ms'), \mathtt{own(b)}\rangle^{Su(\mathtt{ts})}}$$

$$\text{FREE:}\quad \frac{\mathtt{b} \in bLoc(H) \quad H' = H \backslash \{(\mathtt{b}, \mathtt{n}, \_, \_)\} \quad \mathtt{n} \geq 0 \quad ms' = ms \backslash \{\mathtt{(b,0)} \mapsto \_, \ldots, \mathtt{(b,n-1)} \mapsto \_\}}{\langle (H, ms), \mathtt{free(b)}\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle (H', ms'), .\rangle^{Su(\mathtt{ts})}}$$

### 4.4.1 Operational semantics for non-atomic raw read and write

The rules for `rawWrite` and `rawRead` are non-atomic write and read that are directly applied to raw pointers. The execution of a raw operation will get stuck under data races. For a location $l$ and $l \mapsto (\mathtt{n}_1, \mathtt{n}_2) \in ms$, data races are defined as (1) $(\mathtt{n}_1 + \mathtt{n}_2 \geq 2) \wedge (\mathtt{n}_2 \geq 1)$, i.e., there are at least two threads accessing the location and at least one of them writes the location. In order to simulate non-atomic operations, both raw read and write are decomposed into two steps that can be interleaved.

The raw write is defined by Rule RawWrite and RawWrite′. It is a two step operation where the first step modifies $ms$ and translates $\mathtt{rawWrite}(l, v)$ to $\mathtt{rawWrite}'(l, v)$ and $ms(l) = (0, 0)$ ensures that the write will not cause a data race. The second step writes $v$ in the heap location $l$, and resets $ms$. The two steps can be interrupted by other threads. The semantics of raw read is similar to the raw write, which is defined by Rule RawRead and RawRead′. Timestamps are not increased, since they are not for safe pointers. We still need the semantics for detecting race conditions. Rule Race- RawRead defines the semantics for the race conditions of the read. It evolves into the stuck state. The race semantics for other operations are similar.

RawWrite:
$$\frac{ms(l) = (0, 0) \quad ms' = ms[l \leftarrow (0, 1)]}{\langle (H, ms), \mathtt{rawWrite}(l, v)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle (H, ms'), \mathtt{rawWrite}'(l, v)\rangle^{\mathtt{ts}}}$$

RawWrite′:
$$\frac{ms(l) = (0, 1) \quad H' = H[l \leftarrow v] \quad ms' = ms[l \leftarrow (0, 0)]}{\langle (H, ms), \mathtt{rawWrite}'(l, v)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle (H', ms'), .\rangle^{\mathtt{ts}}}$$

RawRead:
$$\frac{ms(l) = (\mathtt{n}, 0) \quad \mathtt{n} \geq 0 \quad ms' = ms[l \leftarrow (\mathtt{n} + 1, 0)]}{\langle (H, ms), \mathtt{rawRead}(l)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle (H, ms'), \mathtt{rawRead}'(l, v)\rangle^{\mathtt{ts}}}$$

RawRead′:
$$\frac{ms(l) = (\mathtt{n}, 0) \quad H(l) = v \quad v \neq \bot \quad \mathtt{n} \geq 1 \quad ms' = ms[l \leftarrow (\mathtt{n} - 1, 0)]}{\langle (H, ms), \mathtt{rawRead}'(l)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle (H, ms'), v\rangle^{\mathtt{ts}}}$$

Race- RawRead:
$$\frac{mem = (S, H, \mathcal{P}, ms) \quad ms(l) = (\mathtt{n}_1, \mathtt{n}_2) \wedge \mathtt{n}_2 > 0}{\langle mem, \mathtt{rawRead}(l)\rangle^{\mathtt{ts}} \rightsquigarrow_m \mathtt{stuck}}$$

Concur- Stuck:
$$\frac{\langle mem, t_1\rangle^{\mathtt{ts}} \rightsquigarrow_m \mathtt{stuck} \text{ or } \langle mem, t_2\rangle^{\mathtt{ts}} \rightsquigarrow_m \mathtt{stuck}}{\langle mem, t_1 \mid\mid t_2\rangle \rightsquigarrow_c \mathtt{stuck}}$$

Concur- 1:
$$\frac{\langle mem, t_1\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem', t\rangle^{\mathtt{ts}}}{\langle mem, t_1 \mid\mid t_2\rangle \rightsquigarrow_c \langle mem', t \mid\mid t_2\rangle}$$

Concur- 2:
$$\frac{\langle mem, t_2\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem', t\rangle^{\mathtt{ts}}}{\langle mem, t_1 \mid\mid t_2\rangle \rightsquigarrow_c \langle mem', t_1 \mid\mid t\rangle}$$

For instance, we consider the concurrent execution of two operations $\mathtt{rawRead}(l)$ and $\mathtt{rawWrite}(l, v)$ under a concurrent semantics based on interleaving, whose grammar is $con ::= tm \mid\mid tm$, where $tm$ can be a memory operation, a value, or "." The semantics is defined by the Rule Concur- 1, Concur- 2, and Concur- Stuck, with the relation $\rightsquigarrow_c$ (The concurrent semantics is only for illustration here. RustSEM concurrent semantics is implemented in CL level.

Assume the initial memory configuration satisfies $ms(l) = (0, 0)$ and $H(l) \neq \bot$, there are 4 possible execution sequences as follows:

(Sequence 1) $\mathtt{rawRead}(l)$; $\mathtt{rawRead}'(l)$; $\mathtt{rawWrite}(l, v)$; $\mathtt{rawWrite}'(l, v)$;

(Sequence 2) $\mathtt{rawWrite}(l, v)$; $\mathtt{rawWrite}'(l, v)$; $\mathtt{rawRead}(l)$; $\mathtt{rawRead}'(l)$;

(Sequence 3) $\texttt{rawRead}(l)$; $\texttt{rawWrite}(l, v)$; $\texttt{stuck}$;
(Sequence 4) $\texttt{rawWrite}(l, v)$; $\texttt{rawRead}(l)$; $\texttt{stuck}$.

Sequences (1) and (2) are safe, but Sequence (3) and (4) are unsafe. For instance, in Sequence (4), after executing $\texttt{rawWrite}(l, v)$, we have that $ms(l) = (0, 1)$, which makes $\texttt{rawRead}(l)$ get stuck (Rule RACE- RAWREAD). We will not present atomic read and write without OBS invariant checking as their semantics rules are trivial.

### 4.4.2 The semantics of safe read and write

The read and write operations for the safe pointers need to maintain the invariants of OBS. The sketch of the semantic rules for $\texttt{read}(p)$ and $\texttt{write}(p)$ is the following: (1) compute the lvalue of $p$, (2) reads value from or writes value to the lvalue of $p$, (3) update the lifetimes of references during the read and write, and finally (4) check the OBS invariants for the resulting memory configuration.

The OBS invariants mapped in memory configurations are defined as follows.

**Definition 8** (*Well-formed memory configurations*) Let $mem = (S, H, \mathcal{P}, ms)$ be a memory configuration. It is well-formed, denoted as $\texttt{wellform}(mem)$, iff it satisfies the following invariants.

1. $\forall \texttt{b}, l, l'.(mem(l) = mem(l') = \texttt{own(b)}) \Longrightarrow l = l'$.
2. $\forall l, l_1, l_2.l_1 \rightarrow_m l \in mem \wedge l_2 \rightarrow_b l \in mem \Longrightarrow \mathcal{L}(l_1, mem) \cap \mathcal{L}(l_2, mem) = \emptyset$.
3. $\forall l, l'.l \rightarrow_s l' \in mem \Longrightarrow \mathcal{P}(l') = (\texttt{ts}, \texttt{ts}') \Longrightarrow \texttt{ts}' \notin \mathcal{L}(l, mem)$.
4. $\forall l, l'.l \rightarrow_m l' \in mem \Longrightarrow \mathcal{P}(l') = (\texttt{ts}, \texttt{ts}') \Longrightarrow \texttt{ts} \notin \mathcal{L}(l, mem) \wedge \texttt{ts}' \notin \mathcal{L}(l, mem)$.

Invariant 1 ensures that no block is owned by more than one location. Invariant 2 ensures that a location cannot be borrowed by two references simultaneously with one of them being mutable. Invariant 3 ensures if a location is borrowed by a shared reference then its latest write timestamp cannot be in the lifetime of the reference. Invariant 4 ensures if a location is borrowed by a mutable reference then both its latest read and write timestamps cannot be in the lifetime of the reference.

Compared with well-formed OBS graphs, well-formed memory configurations also contain *unique owner* and *lifetime disjoint* invariants, but no *lifetime inclusion* invariant. Lifetime inclusion invariant will be maintained by the semantic rules directly. Invariants 3 and 4 of well-formed memory configurations correspond to the permission functions, but not exactly since it only disable the latest read and write. We will prove that it is enough to enforces memory operations to follow the permission functions later in Sect. 4.5.

LV- DEREF:
$$\frac{\begin{array}{c}(1)\ \langle mem, \texttt{lv}(p)\rangle^{\texttt{ts}} \rightsquigarrow_m \langle mem_1, (l, wp_1)\rangle^{\texttt{ts}} \\ (2)\ [\![mem_1]\!].ms(l) = (\texttt{n}, 0) \wedge \texttt{n} \geq 0 \\ (3)\ mem_1(l) = \texttt{ref}(lt, p')\quad (4)\ mem_2 = \texttt{extLT}(mem_1, l, \texttt{ts}) \\ (5)\ \langle mem_2, \texttt{lv}(p')\rangle^{\texttt{ts}} \rightsquigarrow_m \langle mem_3, (l', wp_2)\rangle^{\texttt{ts}} \\ (6)\ \texttt{wellform}(mem_3) \\ (7)\ wp = (mem_1(l) = \texttt{mut}(lt, p')?true : false)\end{array}}{\langle mem, \texttt{lv}(*p)\rangle^{\texttt{ts}} \rightsquigarrow_m \langle mem_3, (l', wp \wedge wp_1 \wedge wp_2)\rangle^{\texttt{ts}}}$$

LV- LOCATION:
$$\langle mem, \texttt{lv}(l)\rangle^{\texttt{ts}} \rightsquigarrow_m \langle mem', (l, true)\rangle^{\texttt{ts}}$$

LV- DEREF:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, wp_1)\rangle^{\mathtt{ts}} \\ (2)\ [\![mem_1]\!].ms(l) = (\mathtt{n}, 0) \wedge \mathtt{n} \geq 0 \quad (3)\ mem_1(l) = \mathtt{ref}(lt, p') \\ (4)\ mem_2 = \mathtt{extLT}(mem_1, l, \mathtt{ts}) \\ (5)\ \langle mem_2, \mathtt{lv}(p')\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_3, (l', wp_2)\rangle^{\mathtt{ts}} \quad (6)\ \mathtt{wellform}(mem_3) \\ (7)\ wp = (mem_1(l) = \mathtt{mut}(lt, p')?true : false)\end{array}}{\langle mem, \mathtt{lv}(*p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_3, (l', wp \wedge wp_1 \wedge wp_2)\rangle^{\mathtt{ts}}}$$

LV- LOCATION:

$$\langle mem, \mathtt{lv}(l)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem', (l, true)\rangle^{\mathtt{ts}}$$

LV- FIELD:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, wp)\rangle^{\mathtt{ts}} \\ (2)\ mem_1(l) = \mathtt{own(b)} \quad (3)\ \mathtt{b} \in bLoc(mem_1)\end{array}}{\langle mem, \mathtt{lv}(p.\mathtt{n})\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, ((\mathtt{b}, \mathtt{n}, ), wp)\rangle^{\mathtt{ts}}}$$

WRITE- REF:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, true)\rangle^{\mathtt{ts}} \\ (2)\ [\![mem_1]\!].ms(l) = (0, 0) \\ (3)\ mem_2 = mem_1[l \leftarrow \mathtt{ref}(\mathtt{ts} \sim \mathtt{ts}, p')] \\ (4)\ mem_3 = \mathtt{addWrite}(mem_2, \mathtt{alias}(p), \mathtt{ts}) \\ (5)\ \mathtt{wellform}(mem_3)\end{array}}{\langle mem, \mathtt{write}(p, \mathtt{ref}(p'))\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_3, .\rangle^{Su(\mathtt{ts})}}$$

WRITE- OWN:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, true)\rangle^{\mathtt{ts}} \\ (2)\ [\![mem_1]\!].ms(l) = (0, 0) \\ (3)\ mem_2 = mem_1[l \leftarrow \mathtt{own}(b)] \\ (4)\ mem_3 = \mathtt{addWrite}(mem_2, \mathtt{alias}(p), \mathtt{ts}) \\ (5)\ \mathtt{wellform}(mem_3) \quad (6)\ \mathtt{b} \in bLoc(mem)\end{array}}{\langle mem, \mathtt{write}(p, \mathtt{own(b)})\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_3, .\rangle^{Su(\mathtt{ts})}}$$

WRITE- PRIMITIVE:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, true)\rangle^{\mathtt{ts}} \\ (2)\ v \in SV \quad (2)\ [\![mem_1]\!].ms(l) = (0, 0) \\ (3)\ mem_2 = mem_1[l \leftarrow v] \\ (4)\ mem_3 = \mathtt{addWrite}(mem_2, \mathtt{alias}(p), \mathtt{ts})\end{array}}{\langle mem, \mathtt{write}(p, v)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_3, .\rangle^{Su(\mathtt{ts})}}$$

READ- REF:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, \_)\rangle^{\mathtt{ts}} \\ (2)\ [\![mem_1]\!].ms(l) = (\mathtt{n}, 0) \wedge \mathtt{n} \geq 0 \quad (3)\ v = mem_1(l) \quad (4)\ v = \mathtt{ref}(lt, p') \\ (5)\ mem_2 = \mathtt{addRead}(mem_1, \mathtt{alias}(p), \mathtt{ts}) \\ (6)\ mem_3 = \mathtt{extLT}(mem_1, l, \mathtt{ts}) \quad (7)\ \mathtt{wellform}(mem_3)\end{array}}{\langle mem, \mathtt{read}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_3, v\rangle^{Su(\mathtt{ts})}}$$

READ- NONREFERENCE:

$$\frac{\begin{array}{c}(1)\ \langle mem, \mathtt{lv}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_1, (l, \_)\rangle^{\mathtt{ts}} \\ (2)\ [\![mem_1]\!].ms(l) = (\mathtt{n}, 0) \wedge \mathtt{n} \geq 0 \quad (3)\ v = mem_1(l) \\ (4)\ v \text{ is not a reference} \quad (5)\ v \neq \bot \\ (7)\ \text{if } v = \mathtt{own}(b) \text{ then } \mathtt{b} \in bLoc(mem) \\ (6)\ mem_2 = \mathtt{addRead}(mem_1, \mathtt{alias}(p), \mathtt{ts})\end{array}}{\langle mem, \mathtt{read}(p)\rangle^{\mathtt{ts}} \rightsquigarrow_m \langle mem_2, v\rangle^{Su(\mathtt{ts})}}$$

***The Semantics of Lvalue***

We first introduce the semantics of Lvalue, as both read and write operations need to use Lvalue. The lifetimes of references accessed during the computation of $\mathtt{lv}$ should be extended since the references are used. The result of $\mathtt{lv}(p)$ is pair $(l, wp)$ where $l$ is the Lvalue and $wp$ is a boolean value indicates whether $p$ is permitted to write.

The first rule for Lvalue is LV- DEREF, which computes the Lvalue of $*p$, where $p$ is a path. We use $\mathtt{ref}(p')$ (resp. $\mathtt{ref}(lt, p')$) to denote a reference, which can be either a shared reference $\mathtt{shr}(p')$ (resp. $\mathtt{shr}(lt, p')$) or a mutable reference $\mathtt{mut}(p')$ (resp. $\mathtt{mut}(lt, p')$).

1. Premise (1) recursively computes the Lvalue of $p$, which is $l$. Lvalue semantics does not increase the timestamp as it is one of the sub-steps for read and write.
2. Premise (2) checks whether the location is being written by a non-atomic operation. $[\![mem_1]\!].ms$ denotes the element $ms$ in the memory configuration $mem_1$. For a location $l$ and $ms(l) = (\mathtt{n}, \mathtt{m})$, $\mathtt{n}$ and $\mathtt{m}$ denotes the number of threads, which are reading and writing the location using raw pointers, respectively. It requires that the number of writes is 0 and the number of reads is greater than or equal to 0. Therefore it could be used to check the data races that a non-atomic write is carrying out but interleaved by safe operations.
3. Premise (3) requires that the location $l$ must be a shared or mutable reference, since we can only dereference a reference.
4. As the reference $l$ is used, its lifetime is extended to the timestamp $\mathtt{ts}$ (Premise (4)) by the function $\mathtt{extLT}$. Assume $mem(l) = \mathtt{ref}(\mathtt{ts}_1 \sim \mathtt{ts}_2, p')$, it is defined as: $\mathtt{extLT}(mem, l, \mathtt{ts}) \triangleq mem[l \leftarrow \mathtt{ref}(\mathtt{ts}_1 \sim \mathtt{ts}, p')]$.
5. Since the referent $p'$ is a path, its Lvalue needs to be further computed (Premise (5)).
6. Premise (6) ensures the memory configuration is still well-formed after the computation.
7. Premise (7) checks whether $l$ is a mutable reference (The notation $e?v_1 : v_2$ denotes that if $e$ is true then $v_1$ otherwise $v_2$).

The resulting permission is decided by $wp \wedge wp_1 \wedge wp_2$, i.e., whether all references accessed by the semantic rule are mutable references. Rule LV- LOCATION and LV- FIELD compute the Lvalues of a location (which is itself) and a field, respectively.

For instance, assume in the memory, we have $\mathtt{s}_1 \mapsto \mathtt{mut}(1 \sim 2, \mathtt{s}_2)$, $\mathtt{s}_2 \mapsto \mathtt{own}(b)$ and $ms(\mathtt{s}_1) = (0, 0)$, $ms(\mathtt{s}_2) = (0, 0)$. If we try to compute $\mathtt{lv}(*\mathtt{s}_1)$, Rule LV- DEREF works as follows.

- Premise (1) computes $\langle mem, \mathtt{lv}(\mathtt{s}_1)\rangle$ and gets $(\mathtt{s}_1, true)$ by Rule LV- LOCATION.
- Premise (2) and (3) are true as $ms(\mathtt{s}_1) = (0, 0)$ and $\mathtt{s}_1$ holds $\mathtt{mut}(1 \sim 2, \mathtt{s}_2)$.
- Premise (4) extends its lifetime to the current timestamp (assume it is 4). Thus value of $\mathtt{s}_1$ is updated as $\mathtt{mut}(1 \sim 4, \mathtt{s}_2)$ now.
- Premise (5) further computes the lvalue of $\mathtt{lv}(\mathtt{s}_2)$, which is $(\mathtt{s}_2, true)$. Therefore $\mathtt{lv}(*\mathtt{s}_1)$ is $\mathtt{s}_2$.
- The write permission is true as $\mathtt{s}_1$ a mutable reference and the write permission of both $\mathtt{lv}(\mathtt{s}_1)$ and $\mathtt{lv}(\mathtt{s}_2)$ are true.

***The Semantics of Write***

The semantics of $\mathtt{write}(p, v)$, writes the value $v$ to the Lvalue of $p$. Rule WRITE- REF defines the semantics for writing a reference value $\mathtt{ref}(p')$ to the Lvalue of the path $p$. The semantics is elaborated as follows.

1. Premise (1) computes the Lvalue of the path $p$ by $\mathtt{lv}(p)$ and requires the permission to write.
2. Premise (2) ensures no read nor write to $l$,
3. Premise (3) writes the value $v$ to the location $l$. The reference value should be attached with the lifetime $\mathtt{ts} \sim \mathtt{ts}$, i.e., the reference starts to have a lifetime.
4. Premise (4) updates the latest write timestamp of $p$'s alias by the function $\mathtt{addWrite}$. Assume $mem = (S, H, \mathcal{P}, ms)$, the function $\mathtt{addWrite}$ is defined as:
   $\mathtt{addWrite}(mem, \mathtt{a}, \mathtt{ts}) \triangleq (S, H, \mathcal{P}[\mathtt{a} \leftarrow (\mathtt{ts}_1, \mathtt{ts})], ms)$ if $\mathcal{P}(\mathtt{a}) = (\mathtt{ts}_1, \mathtt{ts}_2)$.
   The function $\mathtt{alias}$ is defined as: $\mathtt{alias}(p) \triangleq \begin{cases} p & p = \mathtt{s} \text{ or } p = \mathtt{s}.\mathtt{n}_1 \ldots \mathtt{n}_m \\ \mathtt{alias}(p') & p = *p' \text{ or } p = (*p').\mathtt{n} \end{cases}$.
   It is the entry alias used by the path $p$.
5. Premise (5) ensures that the memory configuration is well-formed after the write.

Rule WRITE- OWN is the semantics for writing an owner to a memory location. Rule WRITE- PRIMITIVE is the semantics for writing primitive values, such as integers.

***The Semantics of Read***

The operation $\mathtt{read}(p)$ reads a value from the Lvalue of a path $p$. READ- REF and READ- NONREFERENCE define the semantics for reading reference and non-reference values, respectively. The function $\mathtt{addRead}$ updates the latest timestamp, at which the alias of $p$ is read, which is similar to $\mathtt{addWrite}$.

### 4.5 Refinement relation between high-level OBS and memory model

In this subsection, we prove the refinement relation between high-level abstraction of OBS and the memory model. We begin with the definition of safe sequences.

**Definition 9** Let $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ be an alternating sequence of memory configurations with timestamps and operations. For each pair $(mem, \mathtt{ts})$, $mem$ is a memory configuration and $\mathtt{ts}$ is the timestamp reaching $mem$. $\pi$ is a safe sequence iff

1. $mem_0$ is an empty configuration, i.e., $S$, $\mathcal{P}$, $ms$ are empty maps and $H$ is an empty set,
2. $op_i$, $1 \le i \le n$, is one of the memory operations: $\mathtt{alloc}(\mathtt{n}, \mathtt{t})$, $\mathtt{read}(p)$, $\mathtt{write}(p, v)$, $\mathtt{free}(\mathtt{b})$,

3. $\langle mem_i, op_{i+1}\rangle^{\mathtt{ts}_i} \rightsquigarrow_m \langle mem_{i+1}, re\rangle^{\mathtt{ts}_{i+1}}$, for all $0 \leq i < n$, where $re$ can only be a value if $\mathtt{op}_{i+1}$ is `read` or `alloc`, otherwise consumed ".".

The memory configurations are the *data refinement* of the high-level OBS graphs. For each memory configuration, it has an underlying OBS graph.

**Definition 10** Let $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ be a safe sequence and $B$ be a block in $mem_i$, $(0 \leq i \leq n)$, whose location is `b`. The OBS graph $(B, V, E, \mathcal{F})$ of $mem_i$ is defined as:

- $V$ is a set of locations in $mem_i$ that can access $B$,
- for any node $a$ in $V$,
$$E(a) = \begin{cases} (o, B) & \text{if } mem(a) = \mathtt{own(b)} \\ (s, a') & \text{if } a \rightarrow_s a' \in mem_i \\ (m, a') & \text{if } a \rightarrow_m a' \in mem_i \end{cases}$$
- $\mathcal{F}(a \rightarrow_b a') = \mathcal{L}(a, mem_i)$. $\mathcal{F}(a \rightarrow_o B) = \mathtt{ts}_j \sim \mathtt{ts}_i$ where for all $j \leq k \leq i$, $mem_k(a) = \mathtt{own(b)}$ and $mem_{j-1}(a) \neq \mathtt{own}(b)$.

It is now sufficient to present Theorem 3, which specifies the refinement relation between the high-level abstraction and the memory model.

**Theorem 3** *Let* $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ *be a safe sequence and B be a block. Assume the OBS graph of B in* $mem_i$ $(0 \leq i \leq n)$ *is* $G = (B, V, E, \mathcal{F})$ *then we have*

1. *(Data refinement) G is well-formed.*
2. *(Operation refinement) For any two nodes* $a, a' \in V \cup \{B\}$ *such that* $\mathcal{F}(a \rightarrow_* a') = lt$, *for any operation* $op_k$ $(0 < k \leq n)$, *which is executed at the timestamp* $\mathtt{ts} \in lt$,
   - *if* $op_k$ *is an operation that reads B by the alias a then* $R_G(a, \mathtt{ts}) = true$,
   - *if* $op_k$ *is an operation that writes B by the alias a then* $W_G(a, \mathtt{ts}) = true$.

The proof is in Appendix C. Theorem 3 shows that all operations by an alias $a$ follow the permission functions of $mem_i$'s OBS graph for $B$.

## 5 The core language and translation semantics

In this section, we present the basic idea of the Core Language (CL) and translation semantics and explain the supported Rust's constructs and the efforts of developing RustSEM. For a Rust program, RustSEM first translates it to a CL program and then invokes the CL semantics to execute the CL program. Because the core of RustSEM, i.e., OBS, has already been formalized in the memory model, in CL semantics, we do not need to be concerned about modeling OBS invariant checkings.

The design of CL is intended to reduce the Rust grammar to a simple core to avoid redundant formalization. The key to the translation semantics from Rust to CL is that the semantics of a Rust program should be preserved in the corresponding CL program.

***CL language***

Fig. 5 illustrates some selected grammar rules of CL. All constructs in CL are expressions $e$ that can be evaluated to values. The notion of variables ($x$), values ($v$), dereferences ($*e$), fields ($e.e'$), arithmetic and boolean expressions, print, skip (`clskip`), and assertions are all

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid *e \mid e.e' \mid \texttt{allocInit}(\texttt{n}, \vec{e}, \texttt{t}) \\
& \mid & e_1 + e_2 \mid e_1 < e_2 \mid e_1 \texttt{\&\&} e_2 \mid \ldots \\
& \mid & \texttt{move}(e) \mid \texttt{\&}e \mid \texttt{\& mut } e \\
& \mid & \texttt{print}(e) \mid \texttt{clskip} \mid \texttt{assert}(e) \\
& \mid & fun \mid branch \mid loop \\
& \mid & concurrency \mid binding \\
& \mid & assign \mid seq \mid mop \\
fun & ::= & \texttt{fun } f\ (\vec{x})\{e\} \\
& \mid & \texttt{call}(f, \vec{e}) \mid \texttt{pcall}(e', f, \vec{e}) \\
branch & ::= & \texttt{case}\{e \rightarrow e'\} \\
loop & ::= & \texttt{loop } e\ \{e'\} \mid \texttt{break} \mid \texttt{continue} \\
thread & ::= & \texttt{fork}(e) \mid \texttt{wait}(e) \\
binding & ::= & \texttt{let } x = e \texttt{ in } \{e'\} \\
assign & ::= & e\texttt{:=}e' \\
seq & ::= & e\texttt{;}e'
\end{array}
$$

**Fig. 5** Selected grammar rules and configurations of CL

standard. The operator `move`($e$) moves the value of an expression. Shared (`&`$e$) and mutable (`& mut` $e$) references are identical to those in Rust.

CL functions are expressions that can be evaluated and assigned to variables. Functions can represent Rust closures, which are anonymous functions with environments. Two function call forms are `call` and `pcall`. The former directly calls the function $f$ with a sequence of arguments, whilst the latter is a polymorphic call `pcall`($e'$, $f$, $\vec{e}$) implementing dynamic dispatches. For instance, for expressions like $e'.f(e_1, \ldots, e_n)$ in Rust, we only know that $e'$ is an object implementing a trait (like an interface) containing $f$. Since there may be more than one type implementing the trait, the concrete function $f$ to be invoked depends on the type of $e'$ at run time. As we model the memory as typed, i.e., each memory location stores values and their types, the type of $e'$ at runtime can be obtained by reading the memory. Branches in CL are defined by the `case` construct. Each case $e \rightarrow e'$ is a guarded action such that $e'$ can be executed only if $e$ evaluates to true. The constructs of threads ($thread$), bindings ($binding$), assignments ($assign$), and sequences ($seq$) are standard. The memory operations $mop$ are also constructs in CL. It means that the memory access in CL invokes the memory operations in Sect. 4. As the OBS is formalized in the memory model, the CL semantics does not need to be concerned about it.

CL language is independent of Rust and can be reused by other programming languages. In the following, we briefly discuss how to reuse CL for C. Most language constructs in CL are general for all programming languages. The only distinguishing constructs are `move` and `&mut`. As C also has the reference operator, mapping C constructs to `move` is modeling direct pointer assignments as `move` operations. But for `&mut`, we have two ways to model it: (1) add user-specified comments to clarify which references are `mute`, (2) do a static analysis over the code to check whether the reference has the potential to be modified in the code.

***The translation from Rust to CL.***

Now let us elaborate on the translation semantics from Rust to CL. The first important topic we need to present in the translation semantics is the type system. Table 1 illustrates a subset of types used in the translation. The types **i32**, **bool**, and **str** are the primitive types for inte-

**Table 1** Main types in *RustSEM*

| | | |
|---|---|---|
| $Type$ | := | **i32** \| **bool** \| **str** \| $CompTy$ \| $PointTy$ \| $FnTy$ \| **trait**$(Id_t)$ |
| $PointTy$ | := | **own**$(Type)$ \| **ref**$(Type)$ |
| $CompTy$ | := | **sumTy**$(Type^+)$ \| **prodTy**$(Types^+)$ |
| $FnTy$ | := | **fnTy**$(Type^+, Type)$ |

**Table 2** Configuration for Rust Level

| | | |
|---|---|---|
| $V$ | $\triangleq$ | $Type \times \mathbb{N}$ |
| $FCtx$ | $\triangleq$ | $Map\langle Id_f, FnTy\rangle$ |
| $CCtx$ | $\triangleq$ | $Map\langle Id_c, CompTy\rangle$ |
| $Tenv$ | $\triangleq$ | $Map\langle Id_v, V\rangle$ |
| $EnvStack$ | $\triangleq$ | $List\langle Tenv\rangle$ |
| $Config_r$ | $\triangleq$ | $\{fCtx : FCtx, cCtx : CCtx, tenv : Tenv, stack : EnvStack\}$ |
| $\mathcal{R}$ | $\in$ | $Config_r$ |

**Table 3** Modeled Rust Features

| Modeled Rust features (37 Features) | | | |
|---|---|---|---|
| • Primitive values | • shared references | • mutable references | • Casting pointers |
| • Arithmetic | • print | • assertion | • Move semantics |
| • Dereference | • field access | • array access | • Binding |
| • Execution block | • assignment | • function | • Impl block |
| • Trait definition | • trait impl | • function call | • Method call |
| • dynamic dispatch | • polymorphism | • closure | • Branch |
| • Pattern matching | • loop | • sequential | • Concurrency |
| • Raw pointers | • intrinsic function | • struct types | • Enum type |
| • Generic types | • Box | • Array | • Vector |

ger, boolean, and string values, respectively. Pointer types include **own**$(Type)$, representing the owner of a resource, and reference type (**ref**(Type)), for references to other variables. Compound types include the sum type **sumTy**$(Type^+)$, representing unions, and the product type **prodTy**$(Type^+)$, representing structs or tuples. The function type **fnTy**$(Type^+, Type)$ contain two elements: the first one is the types for parameters and the second one is the return type. **trait**$(Id_t)$ denotes the types of traits, where $Id_t$ represents the name of the trait. In our semantics, traits are considered as types.

The translation semantics translates a Rust program to a CL program. Table 3 shows the constructs in Rust that can be translated into corresponding CL constructs. The translation semantics covers all the important features related to operational semantics, which enables RustSEM to run real Rust programs.

The configuration for the semantics of this level is shown in Table. 2. $V \triangleq Type \times \mathbb{N}$ is used to record the information of variables, which has two elements: (1) the type of the variable and (2) the lifetime, which is a natural number. The notation $Tenv$ (type environment) is a type of maps from variable identifiers to their type and lifetime. The configuration keeps

context types for functions and compound types that are necessary to check the types of function identifiers and compound type identifiers. This information is stored in types $FCtx$ and $CCtx$, respectively mapping function and compound identifiers to function types and compound types.

It is possible for a program to declare bindings of variables with the same name but different types in nested scopes. When this happen, the type environment of the outer scope must be recovered after the inner scope finishes. This is performed by introducing a stack $EnvStack$ for type environments. The translation semantics pushes the current environment in the stack when a new scope is created, and will recover the top of the stack when the scope finishes. In this way, any possible type overwriting in the inner scope will be removed.

Finally, $Config_r$ is the type of the configuration, containing: function and compound types definitions, the type environment and its stack.

The translation semantics is defined by the relation $(Config_r \times \mathbb{N} \times Rust_e) \times (Config_r \times \mathbb{N} \times CL_e)$, notated by $R_c \mid_{lt} E_R \rightarrow_r R'_c \mid_{lt'} E_{cl}$, indicating that given an initial configuration $R_c$ and a lifetime $lt$, the semantics translates the Rust program $E_R$ to the CL expression $E_{cl}$ with the new configuration $R'_c$ and lifetime $lt'$. Giving $R = (fCtx, cCtx, tenv, stack)$ with $R \in Config_r$, $R.fCtx = fCtx$, $R.cCtx = cCtx$, $R.tenv = tenv$, and $R.stack = stack$.

We present our translation semantics for variable bindings, functions, and traits in the following.

***Variable Bindings in Rust.***

A variable binding in Rust creates a new variable and binds it to the value of an expression. The translation from a binding to CL constructs needs to know the lifetime of the binding. Therefore, the translation cannot be done by only scanning the binding. It also needs to scan the whole scope of the variable before finishing the translation.

BINDING- NONCOPYABLE:

$$\frac{\begin{array}{c}(1)\ e : T \quad T \text{ is not copyable} \quad T \text{ implements the } Drop \text{ trait} \\ (2)\ \mathcal{R} \mid_{lt} e \rightarrow_r^+ \mathcal{R}' \mid_{lt} e_{cl} \quad (4)\ \mathcal{R}'' \mid_{lt} stmts \rightarrow_r^+ \mathcal{R}''' \mid_{lt} S \\ (3) tenv' = \mathcal{R}'.tenv \cup (x \mapsto (T, tl)) \quad \mathcal{R}'' = \mathcal{R}'(vInfo/vInfo', tenv/tenv')\end{array}}{\begin{array}{c}\mathcal{R} \mid_{lt} let\ x = e; stmts \rightarrow_r \mathcal{R}''' \mid_{lt} let\ x = \\ \textbf{\#move}(e_{cl})\ in\ \{S;\ pcall(x, drop, .args)\}\end{array}}$$

Rule BINDING- NONCOPYABLE defines the translation semantics of a variable binding from a non-copyable expression $e$ of a type implementing the Drop trait to the variable $x$ (Premise 1). The rule translates $let\ x = e$, into a CL binding, where $e_{cl}$ is a CL expression translated from $e$ (Premise 2). The expression $e_{cl}$ is translated first with the initial configuration $\mathcal{R}$, resulting in a configuration $\mathcal{R}'$. $\mathcal{R}'$ is updated with the information of the binding, storing the type and lifetime of $x$ in the environment $tenv$ (Premise 3). The type is $T$ inherited from $e$ and the lifetime is $lt$.

In Rust, non-copyable expressions must do a ownership transfer to the assigned value, therefore the CL binding must move the ownership from $e_{cl}$. The translation must also consider that $x$ is only available in the statements $stmts$ within its scope. Therefore, the translation from $stmts$ to its CL expression $S$, should use the configuration $R''$, which stores the information of $x$ (Premise 4). Finally, since $T$ implements the trait $Drop$, the call to the drop function of $x$ should be executed after the execution of $S$.

***Traits and Functions.***

Traits in Rust are like interfaces in Java or some other object-oriented programming languages. The provide are a way to implement polymorphism and to tell the Rust compiler about the functionality that a type must implement.

CL supports dynamic dispatch with the support of typed memory model greatly easing the translation from Rust to CL.

$$\frac{\begin{array}{c}\textsc{Trait- Definition}\ : \\ (1)\ \forall 1 \le i \le k, \exists n, FN_i = \textbf{fn}\ f_i(x_{i1} : t_{i1}, \ldots, x_{in} : t_{in})\text{->}t_i \\ (2)\ fCtx' = fCtx \cup \{Id_t(T, f_i) \mapsto \textbf{fnTy}(t_{i1}, \ldots, t_{in}; t_i)\}\end{array}}{\mathcal{R} \mid_{lt} trait\ T\{FN_1 \ldots FN_k\} \rightarrow_r \mathcal{R}(fCtx/fCtx') \mid_{lt} Skip}$$

Rule Trait- Definition defines the translation semantics for trait definitions in Rust. The symbol $Id_t$ creates a function identifier combining a trait identifier with a function identifier. For each function declaration with $n$ argumets $\textbf{fn}\ f_i(x_{i1} : t_{i1}, \ldots, x_{in} : t_{in})\text{->}t_i$ in a trait $T$ (Premise 1), the pair $Id_t(T, f_i) \mapsto fnTy(t_{i1}, \ldots, t_{in}; t_i)$ is added to the type context (Premise 2). This characterizes the function type of the method $f_i$ in the trait $T$. In order to make the rule easy to explain, here we only give a simplified version function declaration, which has no lifetime variables and other type parameters.

$$\frac{\begin{array}{c}\textsc{Type- Trait- Impl}\ : \\ \forall 1 \le i \le k, \exists n, FN_i = \textbf{fn}\ f_i(x_{i1} : t_{i1}, \ldots, x_{in} : t_{in})\text{->}t_i\ \{stmts_i\} \\ (1) FN_i' = \\ \textbf{fn}\ Id_f(T, f_i)(x_{i1} : replace(t_{i1}, T), \ldots, x_{in} : replace(t_{in}, T))\text{->}\ t_i\{stmts_i\} \\ (2) \mathcal{R} \mid_{lt} FN_1' \ldots FN_k' \rightarrow_r^+ \mathcal{R}' \mid_{lt} CL_P\end{array}}{\mathcal{R} \mid_{lt} impl\ tt\ for\ T\{FN_1 \ldots FN_k\} \rightarrow_r \mathcal{R}' \mid_{lt} CL_P}$$

Rule Type- Trait- Impl defines the translation semantics of the implementation of the trait $tt$ for the type $T$. Each function definition $\textbf{fn}\ f_i(x_{i1} : t_{i1}, \ldots, x_{in} : t_{in})\text{->}t_i\{stmts_i\}$ in the trait $tt$ is translated to

$$\textbf{fn}\ Id_f(T, f_i)(x_{i1} : replace(t_{i1}, T), \ldots, x_{in} : replace(t_{in}, T))\text{->}t_i\{stmts\}$$

The notation $Id_f(T, f_i)$ creates a function identifier that combines the type implementing the trait, the trait and the function implemented by the type. Functions in traits may use $self$ in types of the input arguments to refer to the type $T$. The type of each parameter is modified by $replace(t_i, T)$, where $replace$ simply replaces $self$ by $T$ in $t_i$ flattening function $f_i$ (Premise 1). The sequence of flattened function definitions $FN_1' \ldots FN_k'$ is translated to CL functions one by one with the rule for general function definition (Premise 2).

Function calls in Rust have different forms. For instance, $f(e_1, \ldots, e_k)$, where $f$ is a function name, calls the function $f$ with the arguments $e_1, \ldots, e_k$. The expression $x.f(e_1, \ldots, e_k)$, where $e$ is an expression and $f$ is a function name, calls the function $f$ of the type of $x$ with the arguments $e_1, \ldots, e_k$. This form of function calls may trigger dynamic dispatch.

$$\frac{\begin{array}{c}\textsc{Type- Fun- Call}: \\ (1)\ \nexists id.T = Trait(id) \quad (2)\ \mathcal{R} \mid_{lt} e \rightarrow_r \mathcal{R}_1 \mid_{lt} e_1' \\ (3) e : T \quad Id_f(T, f) \mapsto fnTy(t_1, \ldots, t_k; t) \in \mathcal{R}_1.fCtx \\ (4)(\texttt{isSelf}\ t_1) \quad (5)\ \forall 1 <= i <= n. \mathcal{R}_i \mid_{lt} e_i \rightarrow_r \mathcal{R}_{i+1} \mid_{lt} e_{i+1}'\end{array}}{\mathcal{R} \mid_{lt} e.f(e_1, \ldots, e_n) \rightarrow_r \mathcal{R}_{n+1} \mid_{lt} call(Id_f(T, f), \texttt{mod\_self}\ e_1'\ t_1, \ldots, e_{n+1}')}$$

$$\frac{\begin{array}{c}\textsc{Type- Fun- PCall}: \\ (1)\ \exists id.T = Trait(id) \quad (2)\ \mathcal{R} \mid_{lt} e \rightarrow_r \mathcal{R}_1 \mid_{lt} e_1' \\ (3)\ e : T \quad (Id_f(T, f) \mapsto fnTy(t_1, \ldots, t_k, ; t)) \in \mathcal{R}_1.fCtx \\ (4)(\neg\texttt{isSelf}\ t_1) \quad (5)\ \forall 1 <= i <= n. \mathcal{R}_i \mid_{lt} e_i \rightarrow_r \mathcal{R}_{i+1} \mid_{lt} e_i'\end{array}}{\mathcal{R} \mid e.f(e_1, \ldots, e_n) \rightarrow_r \mathcal{R}_{n+1} \mid pcall(e_1', f, e_2', \ldots, e_{n+1}')}$$

We illustrate the translation semantics for function calls $e.f(e_1, \ldots, e_k)$, where $e$ is an expression of type $T$ and $e_1 \ldots, e_k$ are arguments, by Rule TYPE- FUN- CALL and TYPE- FUN- PCALL. We assume that a type does not implement two functions with the same function name.

In Rule TYPE- FUN- CALL, we assume the expression $e$ is of type $T$ and it is not a trait (Premise 1). Therefore, the version of the function definition for the identifier $Id_f(T, f)$ can be decided at compilation time, and the Rust call is translated into a static CL call. On the other hand, in Rule TYPE- FUN- PCALL, the type of $e$ can only be inferred as a trait (Premise 1), for which we do not know during translation which version of the functions needs to be executed, but only the function type, invoking a polymorphic call $pcall$.

In both cases, the argument $e$ is translated to $e'_1$ (premise 2). The function type of $Id_f(T, f)$ is obtained from $fCtx$ (premise 3). The first argument of a function for a type can be "$self$", or a reference to "$self$" (Premise 4). In such case, $e'_1$ should be added as the first argument after modifying it by `mod_self` $e'_1$ $t_1$. The term `mod_self` modifies the CL expression $e'_1$ according to the type $t$ to obtain the CL reference of $e'_1$ if $t$ is a reference of "$self$". Finally, input parameters $e_i$ are translated into $e'_i$ (Premise 5). Rule TYPE- FUN- PCALL shows the case where the first argument does not refer to *self*.

Developing a formal semantics for real-world languages from scratch always requires huge efforts. RustSEM is formalized in $\mathbb{K}$ and consists of around 1100 semantic rules, taking two and a half man-years.

## 6 Evaluation on correctness and applicability to verification

In this section, we evaluate the correctness of our RustSEM (Sect. 6.1) and its applicability to runtime verification (Sect. 6.2) and formal verification (Sect. 6.3). All experiments are conducted on a computer with an Intel Xeon(R) E5-1650-v3 CPU at 3.50GHz $\times$ 12 and a 16GB DDR4 RAM.

### 6.1 The correctness of the semantics

We first formally define the notions of semantics and its correctness. Then, we will show the correctness of our semantics by testing its consistency (i.e., absence of ambiguities), functionality correctness (compared with the Rust compiler), and OBS invariant correctness in detecting memory errors.

**Definition 11** (*Semantic rules and semantics*) Let $C$ be a set of configurations and $M$ be a set of terms, where a term could be a program, a statement, or a value. A *semantic rule* is a partial function: $Ru : C \times M \rightharpoonup C \times M$, i.e., mapping (or reducing) a pair $(c, m)$ of configuration $c$ and term $m$ to a new pair $(c', m')$ of configuration $c'$ and term $m'$. A semantic rule is also denoted by $(c, m) \rightsquigarrow (c', m')$, where $c, c'$ are configurations and $m, m'$ are terms. A *semantics* is a set of semantic rules: $Sem = \{Ru_0, \ldots, Ru_n\}$.

**Definition 12** (*Execution traces of semantics*) Let $Sem$ be a semantics. A finite execution trace of $Sem$ is a sequence $\pi = (c_0, m_0)(c_1, m_1) \ldots (c_n, m_n)$, such that for each $0 \leq i < n$, $\exists Ru \in Sem$ such that $Ru(c_i, m_i) = (c_{i+1}, m_{i+1})$. The trace $\pi$ is called terminating if for all rule $Ru \in Sem$, $Ru((c_n, m_n))$ is undefined. An infinite or divergent trace is an infinite sequence $\pi = (c_0, m_0)(c_1, m_1) \ldots$, such that for each $0 \leq i$, $\exists Ru \in Sem$ such

that $Ru(c_i, m_i) = (c_{i+1}, m_{i+1})$. We denote by $Trace_{Sem}(c, m)$ the set of all terminating or infinite traces of $Sem$ with $(c, m)$ as the initial pair.

**Definition 13** (*Correctness of semantics*) Let $Sem$ be the semantics of a language, $P$ be a program written in the language, $pre_P \subseteq C$ (a subset of configurations) be the precondition of $P$ and $post_P \subseteq C \times (C \cup \{\infty\})$ be the postcondition of $P$, where $\infty$ denotes that the computation is divergent.

The semantics $Sem$ is called *partially correct* iff, for any program $P$ and any configuration $c \in pre_P$, there exists either a terminating trace $(c, m) \dots (c', m')$ satisfying $((c, m), (c', m')) \in post_P$ or an infinite trace starting from $(c, m)$ satisfying $((c, m), \infty) \in post_P$.

The semantics $Sem$ is called *completely correct* iff, for any program $P$ and any configuration $c \in pre_P$, any terminating trace $(c, m) \dots (c', m')$ satisfies $((c, m), (c', m')) \in post_P$ and any infinite trace starting from $(c, m)$ satisfies $((c, m), \infty) \in post_P$.

We have used over 400 test programs for testing the memory operations and the CL semantics and over 300 test programs for the translation semantics. These test programs mainly come from the Rust benchmark [11], the Rust libraries, and the Rust textbook [12]. Besides, we have also developed new test programs for testing our new grammar for memory operations and CL.

**Ambiguity testing.** The first correctness criterion is the *consistency* of semantics, i.e., the absence of ambiguities (no more than one semantic rule can be applied to the same pair of configuration and term). An ambiguous semantics may be *partially correct* but is not *completely correct*. Thus, we must detect ambiguities to ensure *complete correctness*. Indeed, partially correct semantics yield an unsound formal verifier.

Let us demonstrate ambiguities by example. Consider the `if-else` construct "`if e` $B_1$ `else` $B_2$" where `e` is an expression and $B_i$ is an execution block for $i = 1, 2$. The semantics of the construct is as follows:

$$\textsc{Branch:}\ \frac{(c, e) \rightsquigarrow (c', v)}{(c,\ \texttt{if}\ e\ B_1\ \texttt{else}\ B_2) \rightsquigarrow (c',\ \texttt{if}\ v\ B_1\ \texttt{else}\ B_2)}$$

$$\textsc{True:}\ \frac{v = \texttt{true}}{(c,\ \texttt{if}\ v\ B_1\ \texttt{else}\ B_2) \rightsquigarrow (c, B_1)}$$

$$\textsc{False:}\ \frac{}{(c,\ \texttt{if}\ v\ B_1\ \texttt{else}\ B_2) \rightsquigarrow (c, B_2)}$$

Rule Branch evaluates the expression $e$ to a value $v$ under the configuration $c$ and $c$ evolves to a new configuration $c'$ since the evaluation may have side effects. Thus the `if-else` construct is reduced to another `if-else` construct with $e$ replaced by its value $v$. Rule True is for the case that $v$ equals `true`, in which the first block should be executed. Rule False is for the case that $v$ equals `false`, in which the second block should be executed. Note that we have deliberately removed the premise $v$=`false` from Rule False, which leads to ambiguity. Indeed, both Rule True and Rule False can match an `if-else` construct with $v$ being `true`.

At the implementation level, the execution engine of $\mathbb{K}$ always selects the first matching rule to execute. That is, if $\mathbb{K}$ first matches Rule True, then the execution is correct. Unfortunately, if Rule False is matched before Rule True, then the execution is incorrect. That is,

the ambiguity results in the inconsistency of the semantics. This bug can be fixed by adding `v=false` as the premise of Rule FALSE.

This example contains another ambiguity. If we do not distinguish expressions (e.g., e) and values (e.g., v), then the construct "`if true` $B_1$ `else` $B_2$ " can match both Rule BRANCH and Rule TRUE, since the value `true` is also an expression. Further, Rule BRANCH can match it infinitely many times, resulting in an incorrect infinite trace. Obviously, such ambiguities should be avoided in our semantics.

Note that the consistency of semantics cannot be automatically checked by $\mathbb{K}$, as $\mathbb{K}$ is not a theorem prover like Coq [15] or Isabelle [16] that can prove given properties of a semantics. As a solution, we propose a new testing strategy that exploits $\mathbb{K}$'s execution engine and verification engine to test our semantics. The following two steps are carried out for each test case.

1. Run $\mathbb{K}$'s execution engine on the test case. This can quickly detect some bugs. However, this is insufficient in detecting ambiguities as the engine always executes the first matching rule in case of ambiguity. That is, formally, the engine selects one of the traces in $Trace_{Sem}(c, m)$.
2. Run $\mathbb{K}$'s verification engine on the test case. This can detect some ambiguities by exploring all possible execution traces. That is, formally, the verification engine searches for all possible execution traces in $Trace_{Sem}(c, m)$.

This testing strategy has detected more than 36 ambiguities, showing that our testing technique is efficient and effective. It is efficient because the test cases are small and the inputs are fixed, thus no state explosion exists. The semantics of one construct in Rust is usually formalized with a collection of $\mathbb{K}$ rules, and thus, one test case can cover a collection of rules. It is effective because it can detect ambiguities, even with simple test cases. For instance, the following test case with the `if-else` construct could detect the two ambiguities in Rules BRANCH, TRUE, and FALSE by running the verification engine.

```
if (true)  {assert(true)} else {assert(false)} if (false) {
assert(false) } else { assert(true)  }
```

**Functionality testing.** We have tested the functional correctness of our semantics by comparing the execution output of every input of every test program with the output of the machine code generated by the Rust compiler version 1.78.0.

Even though we compared with a specific version of the Rust compiler, our initial objective is to separate Rust's constructs from CL. CL holds all the ownership and borrowing features. Rust's constructs may evolve in the future, but our semantics does not need to modify CL; instead, only the translation from Rust's new constructs to CL needs to be modified.

**OBS invariant testing.** We have tested the correctness of OBS formalization and comparing the results with the compilation result of the Rust compiler. The experimental results show that:

(1) Some programs are accepted by RustSEM but rejected by the Rust compiler due to its over-approximation of lifetimes to improve the efficiency of its OBS analysis. For instance, Listing 9 shows a function that fails to pass the Rust compiler. It is a function with one parameter `y`, which refers to an integer value. The Rust compiler always requires the lifetime of the reference (the parameter `y`) passed to the function `foo` to outlive the function body. As `z` is only available inside `foo`, after the call to `foo`, the Rust compiler reports that `y` points to an invalid memory location. However, the value of `y` is not returned by `foo`. Therefore it cannot be used by the caller of `foo`, and we do not need to enforce its lifetime to outlive the function body of `foo`. Therefore, the program has no memory error. It is understandable for

the Rust compiler to enforce such restrictions, since it makes the OBS checking of the Rust compiler very efficiently. But RustSEM aims at providing a more accurate understanding of OBS and thus relaxes such restrictions.

(2) Some programs are rejected by RustSEM but accepted by the Rust compiler. Most of those programs are mixed with safe and unsafe operations.

For instance, the program in Listing 8 contains a dangling pointer. Indeed, Line 8.1 creates a vector `vec` and assigns it to the variable `p` (called *owner* in Rust). Line 8.2 creates a raw pointer `r` (an unsafe pointer) to `p`. Line 8.3 moves the value of `p` to `y`. The variable `p` will lose ownership of the vector after the *move* action; therefore, `p` should not access the vector hereafter. Line 8.4 tries to access the vector via the raw pointer `r` and implicitly via `p` (as `*r` accesses `p`), resulting in a dangling pointer error. In stacked borrows [4], this program is also identified as an undefined behavior. Stacked borrows also give a model of OBS using dynamic OBS checking, but stacked borrows strictly follow Rust's type system. This means that the OBS semantics in RustSEM and stacked borrows are Different. For instance, Listing 9 is an example that also cannot pass stacked borrows' OBS checking.

**Listing 8** A Dangling Pointer Example

```
8.1 let p = vec![1,2];
8.2 let r = & p as *const Vec<i32>;
8.3 let y = p;
8.4 println!("{}", unsafe{(*r)[1]});
```

**Listing 9** An Example Rejected by Stacked Borrows

```
9.1 fn foo(mut y:&i32){
9.2     let z = 1;
9.3     y = & z;
9.4 }
```

### 6.2 Runtime verification

RustSEM provides a runtime checker for detecting memory errors, since the memory model integrates a checking mechanism for detecting invalid memory accesses (c.f. Sect. 4). RustSEM can detect memory errors: accessing uninitialized locations, reading dangling pointers, double frees, data races, and deadlocks, ownership and borrowing errors, and buffer overflows. Our runtime checker is implemented using a semantics instrumentation technique rather than the source code instrumentation techniques used by most existing runtime checkers such as Chen et al. [17].

Table 4 shows the results for runtime verification. For each type of memory error, we collected a set of test programs with several inputs that can trigger the errors. RustSEM can successfully detect all the memory errors in the test programs with memory errors, Also, note that data races are not always detected in every execution since the executions are non-deterministic.

Compared with other runtime checkers, RustSEM is slower and depends on the efficiency of $\mathbb{K}$'s execution engine. Note that, to execute a program, $\mathbb{K}$ first constructs its abstract syntax

**Table 4** Results for runtime verification

| Memory errors | TC | IN | Time | Mem |
|---|---|---|---|---|
| Read uninit. values | 22 | 10 | 6.16 | 387,240 |
| Read dangling pointers | 11 | 10 | 5.89 | 390,356 |
| Double frees | 10 | 10 | 5.73 | 346,096 |
| Data races, deadlocks | 14 | 10 | 7.36 | 394,152 |
| Ownership and borrowing | 40 | 10 | 6.64 | 344,088 |
| Buffer overflows | 21 | 10 | 7.16 | 379,788 |

In the table head, "TC", "IN", "Time" and "Mem" denote the number of test cases, the number of inputs for each test case, average execution time in seconds and average memory consumption in KB, respectively

tree and searches for the corresponding semantic rules for the tree at each execution step. Thus, the time spent parsing and interpreting the source code leads to slower execution.

There is another method to construct a runtime checker for Rust based on semantics, which first generates the binary code of Rust programs based on the semantics and then runs the binary code. One of such case is the C semantics in K-Framework [18]. One of our future works is to apply this method to our Rust semantics.

One advantage of RustSEM is that the runtime verification is based on semantics instrumentation instead of code instrumentation. To the best of our knowledge, RustSEM is the first runtime verification tool capable of checking the ownership and borrowing constraints in Rust programs.

### 6.3 Formal verification for both memory and functional properties

$\mathbb{K}$ has a verification engine, which can be used to construct a program verifier for a language by instantiating it with the operational semantics of that language. $\mathbb{K}$'s verification engine takes an operational semantics given in $\mathbb{K}$ and generates queries to a theorem prover (for example, Z3 [19]). The program correctness properties are given as reachability rules between matching patterns in $\mathbb{K}$ semantic rules. Internally, the verifier uses the operational semantics to perform symbolic execution. Also, it has an internal matching logic prover for reasoning about implications between patterns (states), which reduces to SMT reasoning (refer to the $\mathbb{K}$ verification infrastructure [20] for more details).

For the verification, it is necessary to specify a pre- and postcondition similar to Hoare logic, and loops must be annotated with invariants. For simplicity, we show the verified properties as a pair $\{Precondition\}\{Postcondition\}$, indicating that if the program's input satisfies the precondition, the result satisfies the postcondition. Thanks to the memory access check formalized in RustSEM, we can uniformly specify memory properties such as data races, buffer overflows, or borrowing errors, as $\{True\}\{\neg stuck\}$, which means that for any input, the program cannot get stuck. For other kinds of properties, such as functional properties, the program should satisfy both the OBS invariants and the properties being specified.

Table 5 shows the verification results of several programs verified in RustSEM using the $\mathbb{K}$ verifier. The program `sum(`$N$`)` is a function that computes the sum from 1 to the input $N$, and the verified property is $\{N \geq 1\}\{N \times (N+1)/2\}$. The program `sumvec(`$V$`)` computes the sum of all elements in $V$, and the verified property is $\{[v_1, \ldots, v_n] \in$ `vec<i32>`$\}\{\Sigma_{1 \leq i \leq n} v_i\}$ where the precondition specifies a vector of $n$ random integers and the postcondition speci-

**Table 5** Results for formal verification

| Programs | | LOC | Time | Mem |
|---|---|---|---|---|
| sum($N$) | | 15 | 44.65 | 698,240 |
| sumvec($V$) | | 18 | 63.92 | 1,470,880 |
| insertion_sort($V$) | | 43 | 125.43 | 8,736,592 |
| polymorphism | | 60 | 42.20 | 678,992 |
| closure | | 22 | 36.45 | 663,268 |
| trait | | 50 | 39.22 | 654,672 |
| concur-race | | 58 | 47.25 | 647,328 |
| concur-order | | 57 | 60.82 | 2,931,760 |
| VecDeque | new | 210 | 76.90 | 1,160,640 |
| | push_front | | 115.33 | 1,667,140 |
| | push_back | | 140.70 | 2,025,244 |
| | reserve | | 85 | 1,859,304 |

In the table head, "LOC", "Time" and "Mem" denote the lines of code, execution time in seconds, and memory consumption in KB, respectively

fies the sum of the $n$ elements. The program `insertion_sort` implements an insertion sort algorithm and the verified property is $\{V \in$ `vec<i32>`$\}\{V' \in$ `vec<i32>` s.t. `sameElem`$(V, V') \wedge$ `order`$(V')\}$ where the precondition specifies an integer vector and the postcondition specifies a new vector with the same elements (checked by `sameElem`) and ensures that the vector is ordered (checked by `order`). The programs `polymorphism`, `closure` and `trait` use polymorphism, closures and traits, respectively, and we have verified some functional properties. The program `concur-race` is a multi-threaded program with a data race bug and the verified property is $\{True\}\{\neg stuck\}$, i.e., the program terminates without getting stuck. The verifier can successfully detect the bug. The program `concur-order` is a multi-threaded program without data races and the verified property is the same as `concur-race`. The verifier explores all possible interleavings in the concurrent program without bugs detected.

The benchmark "`VecDeque`" is a module from the Rust standard library. It is a library implementing a ring buffer in which an element can be pushed to its head or tail. One of its previous versions contains a bug [21]. `VecDeque` is a non-trivial standard library, as it implements more than 15 methods and invokes other low-level Rust libraries such as `ptr` and `RawVec`. Here, we only illustrate the verification of methods `new`, `push_front`, `push_back`, and `reserve`. The method `new` creates a new `VecDeque`. The methods `push_front` and `push_back` push a new element to the head and the tail of a `VecDeque`, respectively. The method `reserve` reserves a memory space of a specific size for the vector. These four methods implement non-trivial functions. For instance, `push_front` tries to push an element to the vector by first checking whether the vector is full. If not full, it just pushes the element and updates its head. Otherwise, it reallocates a memory space, copies elements to the new memory space recomputes the vector's head and tail and finally pushes the element.

The property to be verified is: $\{0 \leq$ `head` $<$ `cap` $\wedge\ 0 \leq$ `tail` $<$ `cap`$\}\{0 \leq$ `head`$' <$ `cap`$' \wedge\ 0 \leq$ `tail`$' <$ `cap`$'\}$. VecDeque has a `head` to indicate the memory location for `push_front` and and a `tail` to indicate the memory location for `push_back`. The variable `cap` is the capacity of the VecDeque. The property requires that the `head` and `tail` of a VecDeque have to be within the range [0, `cap`) before and after executing any of

its methods. The primed variables denote the variables' values after executing the methods. The bug [21] can be rediscovered by RustSEM in the method `reserve`, resulting in a buffer overflow. RustSEM does not support infinite heap structure specification and verification now. For the verification, we use an approach similar to bounded model checking, setting a bound in the maximum capacity of `vecDeque` to 16, limiting the state space exploration.

Now, we discuss the limitations of formal verification. Firstly, the property specification language in $\mathbb{K}$ needs to specify pre- and post-conditions and loop invariants for each memory location, which requires much effort. We only give an abstract description of the properties being verified here. More details of specification languages could be found in $\mathbb{K}$ verification infrastructure [20]. Secondly, we cannot specify infinite heap data structures like trees. $\mathbb{K}$ has the capability to reason about infinite heap data structures, such as binary search trees, which is shown in $\mathbb{K}$ verification infrastructure [20]. The challenge is that $\mathbb{K}$ verification infrastructure uses a memory model different from RustSEM, as RustSEM incorporates OBS. Therefore, we cannot directly inherit the framework from it. More work needs to be done to define a language that could be used to define infinite data structures in RustSEM inductively. One solution is discovering the relations between RustSEM's memory model and separation logic [22][23]. In this paper, we focus on the semantics of Rust. A more user-friendly and powerful verifier is our future work.

## 7 Related work

**Rust semantics** Table 6 compares the existing semantics for Rust on their target languages (lang), their formalization of OBS (language-level OBS or memory-level OBS), as well as their verification capabilities, i.e., whether they support automatic verification (AV) and machine-checked manual verification (MV).

Except for RustSEM, all other semantics works model OBS at the language level. RustBelt, Oxide, and Patina formalize either a variant or an outdated version of Rust. RustHornBelt combines RustBelt and RustHorn [24] to extend RustBelt with the verification ability for the interaction between safe and unsafe constructs. However, it still uses a type system instead of a memory-level operational semantics. KRust gives semantics directly on the Rust grammar, but it covers a limited subset. For instance, they do not model traits, closures, concurrency, and pattern matching. RustSEM covers a much larger set of Rust's constructs than KRust.

For the verification, RustSEM and KRust can be used for automated verification. RustBelt, implemented in the Coq theorem prover, can verify $\lambda_{Rust}$ programs using Coq; the verification is interactive and needs expertise and manual inspection. Patina and Oxide have yet to be implemented in any tool. Thus, they provide neither automated nor machine-checked verification.

**Rust verification** CRust [25], SMACK-Rust [26] and Viper-based Prusti [27] can verify Rust programs by translating them into the input languages of existing verification tools, instead of building formal semantics for Rust. For instance, SMACK-Rust compiles Rust programs into the LLVM code which can be verified by SMACK [28]. Prusti translates a subset of safe constructs to the intermediate language of Viper [29] to construct core proofs. Aeneas [30] is an interesting work to leverage Rust's type system to eliminate memory reasoning for safe Rust programs and instead focus on Rust programs' functionalities.

Stacked borrows [4] presents an alias model to regulate unsafe pointers in Rust by proposing a dynamic OBS checking. The differences between stacked borrows and our work include:

**Table 6** A comparison of semantics works

| | Language | Language-level OBS | Memory-level OBS | AV | MV |
|---|---|---|---|---|---|
| RustBelt [2] | $\lambda_{Rust}$ | ✓ | | | ✓ |
| RustHornBelt [38] | $\lambda_{Rust}$ | ✓ | | | ✓ |
| Patina [3] | Old Rust | ✓ | | | |
| Oxide [39] | Oxide | ✓ | | | |
| KRust [40] | Rust | ✓ | | ✓ | |
| RustSEM | Rust | | ✓ | ✓ | |

(1) Stacked borrows formalize OBS at the language level, whilst our work formalizes OBS at the memory level. (2) Stacked borrows strictly follow Rust's type system, whilst our work relaxes some restrictions of Rust's type system to yield a more accurate OBS semantics.

**Ownership and borrowing** The concept of ownership has been proposed for many years and most related works exploit type systems to enforce various ownership disciplines. Cyclone [31] is designed to be a safe dialect of the C language by using region based memory management [32]. Mezzo [33] and Alms [34] follow the ML tradition and employ substructural type systems for managing ownership. There are also many type systems associating read and write permissions to aliases [35], such as Mezzo [33], Pony [36] and AEminium [37]. All these works have different technique details with Rust and exploit type systems to enforce memory safety. Our work aims to explain Rust's techniques from an operational aspect at the memory level to provide new insight into OBS.

## 8 Conclusion

We have proposed a memory model integrating of OBS checking and an executable operational semantics for Rust. Compared with existing works, it covers a larger subset of the major language constructs of Rust. The core of the semantics is the memory model with the operational semantics of OBS, which other languages can potentially reuse to improve memory safety. We have proved the refinement relation between OBS's high-level abstraction and the memory model's operational semantics. Moreover, we proposed a technique for testing semantic consistency to detect ambiguities and provide completely correct semantics. We have shown that RustSEM can be applied in automated runtime and formal verification.

In future works, we are working on reusing our memory model for the $\mathbb{K}$ semantics of the C programming language to improve C memory safety by checking the preservation of the OBS invariants. Moreover, we are also working on developing a more efficient Rust verifier based on the semantics.

## Appendix: A Proof of Lemma 1

***Proof*** Assume there is a cycle in a well-formed OBS graph, there must exist $a \rightarrow_b a'$ and $a' \rightarrow_b a_1 \ldots a_2 \rightarrow_b a$. We can infer that $\mathcal{F}(a \rightarrow_b a') \subset \mathcal{F}(a_2 \rightarrow_b a)$ and $\mathcal{F}(a_2 \rightarrow_b a) \subset \mathcal{F}(a \rightarrow_b a')$ by the condition (2) of Definition 6, which cannot be true, □

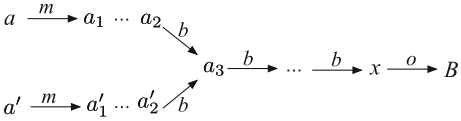



**Fig. 6** The proof idea of Theorem 1

## Appendix: B Proof of Theorem 1 and 2

***The proof of Theorem 1*** This theorem specifies that exclusive mutation guarantee are maintained in a well-formed OBS graph with respect to the permission functions.

If the owner and all references are shared aliases then the theorem is trivially proved. If there is a mutable alias enabled at the timestamp $t$ then it is proved by two cases.

*Case 1:* Assume the owner is the mutable alias. We can infer that there is no reference that has the write or read permission to the memory location at $t$. Otherwise the reference should disable the owner according to the definition of the permission functions and the unique owner invariant of well-formed OBS graphs.

*Case 2:* Assume there is a mutable reference $a$ to the memory block enabled at $t$, and the owner of the memory block is $x$, we have that there exists a link $a \rightarrow_m a_1 \ldots x \rightarrow_o B$. The owner and all other references in the link should be disabled by $a$ and thus their write permissions are not enabled.

Now assume there is another reference $a' \neq a$ enabled at $t$. It should not be in the link from $a$ to $x$. We have that there exists another link $a' \rightarrow_m a'_1 \ldots x \rightarrow_o B$. Then there must exists $a_2, a'_2, a_3$ such that $a \rightarrow_m a_1 \ldots a_2 \rightarrow_b a_3 \ldots x \rightarrow_o B$ and $a' \rightarrow_m a'_1 \ldots a'_2 \rightarrow_b a_3 \ldots x \rightarrow_o B$ (See Fig. 6). According to Invariant (3) of the definition for well-formed OBS graphs (Definition 6), the lifetimes of $a_2 \rightarrow_b a_3$ and $a'_2 \rightarrow_b a_3$ should not intersect. According to Invariant (2) of Definition 6, the lifetime of $a \rightarrow_m a_1$ should be within the lifetime of $a_2 \rightarrow_b a_3$ and the lifetime of $a' \rightarrow_m a'_1$ should be within the lifetime of $a'_2 \rightarrow_b a_3$. Thus the lifetimes of $a \rightarrow_m a_1$ and $a' \rightarrow_m a'_1$ cannot intersect. Therefore the lifetime of $a' \rightarrow_m a'_1$ does not contain the current timestamp $t$ and $a'$ is not permitted to write. □

***The proof of Theorem 2*** It can be proved by the definition of permission functions $R_G$ and $W_G$ that both requires $t$ is in the lifetime of the alias $a$, and the second condition (lifetime inclusion invariant) in Definition 6. □

## Appendix: C Proof of Theorem 3

The first claim that $G$ is a well-formed OBS graph is proved by Lemmas 2 and 4. Lemma 2 proves that each memory configuration in a safe sequence is a well-formed configuration and Lemma 4 proves the lifetime inclusion variant for well-formed configurations. The second claim is proved by Lemmas 5 and 6.

**Lemma 2** *Let* $\pi = (mem_0, ts_0), op_1, (mem_1, ts_1), \ldots, (mem_{n-1}, ts_{n-1}), op_n, (mem_n, ts_n)$ *be a safe sequence. Then all* $mem_i, 0 \leq i \leq n$, *are well-formed.*

***Proof*** Since $m_0$ is an empty configuration, it is trivially well-formed. The allocation operation does not change the well-formedness of a memory layout. The free operation also does not change the well-formedness of a memory configuration, since it only removes a block from the

heap. For read and write operations, the semantic rules always ensure the resulting memory configurations are well-formed. □

**Lemma 3** *Let* $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ *be a safe sequence. Let* $mem_i, 0 \leq i \leq n$ *be a memory configuration.. We have:*

1. *For any* $\mathtt{a} \rightarrow_s \mathtt{a}' \in mem_i$*, there is no write by* $\mathtt{a}'$ *during the lifetime of* $\mathtt{a}$ *in* $mem_i$*.*
2. *For any* $\mathtt{a} \rightarrow_m \mathtt{a}' \in mem_i$*, there is no read nor write by* $\mathtt{a}'$ *during the lifetime of* $\mathtt{a}$ *in* $mem_i$*.*

***Proof*** We first prove the claim (1). Assume $\mathtt{a}$'s lifetime is $\mathtt{ts}_b \sim \mathtt{ts}_e$ in $mem_i$ (i.e., $\mathcal{L}(\mathtt{a}, mem_i) = \mathtt{ts}_b \sim \mathtt{ts}_e$) and $\mathtt{ts}$ is a timestamp at which $\mathtt{a}'$ is used to write and $\mathtt{ts} \in \mathtt{ts}_b \sim \mathtt{ts}_e$.

The write by $\mathtt{a}'$ at $\mathtt{ts}$ should not be the latest write in $mem_i$, that is, if $mem_i = (S, H, \mathcal{P}, ms)$ and $\mathcal{P}(\mathtt{a}') = (\_, \mathtt{ts}')$ then $\mathtt{ts} \neq \mathtt{ts}'$, otherwise it should be disabled according to Definition 8. Therefore we have $\mathtt{ts}' > \mathtt{ts}_e$.

Without loss of generality, we assume there is no write by $\mathtt{a}'$ during $\mathtt{ts} \sim \mathtt{ts}_e$. Then we have that the memory configuration $mem_k$ in $(mem_k, \mathtt{ts}_k)$ with $\mathtt{ts}_k = \mathtt{ts}_e$ is not a well-formed memory configuration as the write by $\mathtt{a}'$ at $\mathtt{ts}$ is the lasting write by $\mathtt{a}'$ in $mem_k$ but the lifetime of $\mathtt{a}$ in $mem_k$ is $\mathtt{ts}_b \sim \mathtt{ts}_e$. The write should be disabled according to Definition 8. This contradicts to Lemma 3. □

**Lemma 4** *Let* $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ *be a safe sequence. Let* $mem_i, 0 \leq i \leq n$ *be a memory configuration and* $B$ *be a block with the location* $b$*. Let the OBS graph of* $B$ *in* $mem_i$ *be* $G = (B, V, E, \mathcal{F})$*. We have that* $G$ *is well-formed.*

***Proof*** From Lemma 2, $mem_i$ is well-formed, which ensures that $G$ satisfies the *unique owner invariant* and *no intersection invariant*. We still need to prove the *lifetime inclusion invariant*.

For references, according the definition of $\mathcal{L}(\mathtt{a}, mem)$, we know that the lifetime of $\mathtt{a}$ will always include the lifetime of $\mathtt{a}'$, where $\mathtt{a}' \rightarrow^b_* \mathtt{a} \in mem$.

We need to consider the owner. If an owner's write permission is always disabled by its references then the ownership cannot be moved. In this case, its lifetime always includes it's references' lifetimes. From Lemma 3, we know that the owner is always disabled. □

**Lemma 5** *Let* $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ *be a safe sequence. Let* $G = (B, V, E, \mathcal{F})$ *be the OBS graph of* $B$ *in* $mem_i, 0 \leq i \leq n$*, we have for any two nodes* $a, a' \in V \cup \{B\}$*, if a read operation* $\mathtt{read}(p)$*, where* $\mathtt{alias}(p) = a$*, is carried out at the timestamp* $\mathtt{ts} \in \mathcal{F}(a \rightarrow_* a')$ *then* $R_G(a, \mathtt{ts}) = true$*.*

***Proof*** The key to prove the lemma is to ensure that during the lifetime of $a$, the read by $a$ at $\mathtt{ts}$ is not disabled. Actually it is proved by Lemma 3. □

**Lemma 6** *Let* $\pi = (mem_0, \mathtt{ts}_0), op_1, (mem_1, \mathtt{ts}_1), \ldots, (mem_{n-1}, \mathtt{ts}_{n-1}), op_n, (mem_n, \mathtt{ts}_n)$ *be a safe sequence. Let* $G = (B, V, E, \mathcal{F})$ *be the OBS graph of* $B$ *in* $mem_i, 0 \leq i \leq n$*, we have for any two nodes* $a, a' \in V \cup \{B\}$*, if a write operation* $\mathtt{write}(p, v)$*, where* $\mathtt{alias}(p) = a$*, is carried out at the timestamp* $\mathtt{ts} \in \mathcal{F}(a \rightarrow_* a')$ *then* $W_G(a, \mathtt{ts}) = true$*.*

***Proof*** Proof is similar to Lemma 5. □

Theorem 3 is proved by Lemmas 2, 4, 5, and 6.

**Acknowledgements** This work was supported in part by the National Natural Science Foundation of China under Grant 62172217 and 61902180, in part by the Joint Research Funds of the National Natural Science Foundation of China and the Civil Aviation Administration of China under Grant U1533130, the National Satellite of Excellence in Trustworthy Software Systems and the Award No. NRF2014NCR-NCR001-30, funded by NRF Singapore under National Cyber-security R&D (NCR) programme.

**Declarations** The datasets generated during and/or analysed during the current study are available in the **Rust-Semantics repository**, https://github.com/MEM-Group/Rust-Semantics.

## References

1. Rust-Team: the Rust language homepage. https://www.rust-lang.org/en-US/ (2016)
2. Jung R, Jourdan J-H, Krebbers R, Dreyer D (2018) RustBelt: Securing the foundations of the Rust programming language. In: Proceedings of the ACM Program. Lang. 2, POPL, Article. https://doi.org/10.1145/3158154
3. Reed E (2015) Patina: a formalization of the Rust programming language. Technical report, University of Washington
4. Jung R, Dang H, Kang J, Dreyer D (2020) Stacked borrows: an aliasing model for rust. Proc ACM Program Lang 4(POPL):41–14132
5. Matsakis N (2016) Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html
6. Roşu G, Şerbănuţă TF (2010) An overview of the K semantic framework. J Logic Algebr Program 79(6):397–434
7. Bogdănaş D, Roşu G (2015) K-Java: a complete semantics of Java. In: Proceedings of the 42nd symposium on principles of programming languages (POPL'15. ACM, New York), pp 445–456. https://doi.org/10.1145/2676726.2676982
8. Hathhorn C, Ellison C, Roşu G (2015) Defining the undefinedness of c. In: Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation (PLDI'15). ACM, New York, pp 336–345. https://doi.org/10.1145/2813885.2737979
9. Ellison C, Rosu G (2012) An executable formal semantics of c with applications. In: Proceedings of the 39th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL'12). ACM, New York, pp. 33–544. https://doi.org/10.1145/2103656.2103719
10. Davidoff SS (2018) How Rust's standard library was vulnerable for years and nobody noticed. https://medium.com/@shnatsel/how-rusts-standard-library-was-vulnerable-for-years-and-nobody-noticed-aebf0503c3d6
11. Rust-Benchmark: Rust Benchmark. https://github.com/rust-lang/rust/tree/master/src/test (2020)
12. Rust-Team (2018) The Rust programming language. Mozilla Research. Mozilla Research. https://doc.rust-lang.org/book/foreword.html
13. Rust-team: non-lexical lifetimes. https://doc.rust-lang.org/edition-guide/rust-2018/ownership-and-lifetimes/non-lexical-lifetimes.html (2018)
14. Matsakis N (2017) Nested method calls via two-phase borrowing. http://smallcultfollowing.com/babysteps/blog/2017/03/01/nested-method-calls-via-two-phase-borrowing/
15. The Coq Proof Assistant. http://coq.inria.fr
16. Nipkow T, Klein G (2014) Concrete Semantics—with Isabelle/HOL. Springer, Heidelberg. https://doi.org/10.1007/978-3-319-10542-0
17. Chen Z, Yan J, Kan S, Qian J, Xue J (2019) Detecting memory errors at runtime with source-level instrumentation. In: Zhang D, Møller A (eds) Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. ACM, New York, pp 341–351. https://doi.org/10.1145/3293882.3330581
18. Team K-F (2024) C-Semantics in KFramework. https://github.com/kframework/c-semantics
19. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J (eds) 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. Lecture Notes in Computer Science. Springer, Heidelberg, vol 4963, pp 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
20. Ştefănescu A, Park D, Yuwen S, Li Y, Roşu G (2016) Semantics-based program verifiers for all languages. In: Proceedings of the 31th conference on object-oriented programming, systems, languages, and applications (OOPSLA'16). ACM, New York, pp 74–91. https://doi.org/10.1145/2983990.2984027
21. issue 44800, V. https://github.com/rust-lang/rust/issues/44800 (2017)

22. Ishtiaq SS, O'Hearn PW (2001) BI as an assertion language for mutable data structures. In: Hankin C, Schmidt D (eds) Conference record of POPL 2001: the 28th ACM SIGPLAN-SIGACT symposium on principles of programming languages, London, UK, January 17–19, 2001. ACM, New York, pp 14–26. http://dl.acm.org/citation.cfm?id=360204
23. Reynolds JC (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, Washington DC, pp 55–74. https://doi.org/10.1109/LICS.2002.1029817
24. Matsushita Y, Tsukada T, Kobayashi N (2021) Rusthorn: CHC-based verification for rust programs. ACM Trans Program Lang Syst 43(4):15–11554. https://doi.org/10.1145/3462205
25. Toman J, Pernsteiner S, Torlak E (2015) Crust: a bounded verifier for Rust (N). In: Cohen MB, Grunske L, Whalen M (eds) 30th IEEE/ACM international conference on automated software engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015. IEEE Computer Society, Washington DC, pp 75–80. https://doi.org/10.1109/ASE.2015.77
26. Baranowski MS, He S, Rakamaric Z (2018) Verifying Rust programs with SMACK. In: Lahiri SK, Wang C (eds) Automated technology for verification and analysis—16th international symposium, ATVA 2018, Los Angeles, CA, USA, October 7–10, 2018, Proceedings. lecture notes in computer science. Springer, Heidelberg, vol 11138, pp 528–535. https://doi.org/10.1007/978-3-030-01090-4. https://doi.org/10.1007/978-3-030-01090-4_32
27. Astrauskas V, Müller P, Poli F, Summers AJ (2019) Leveraging Rust types for modular specification and verification. In: To appear in object-oriented programming systems, languages, and applications (OOPSLA). ACM, New York. https://doi.org/10.1145/3360573
28. Carter M, He S, Whitaker J, Rakamaric Z, Emmi M (2016) SMACK software verification toolchain. In: Dillon LK, Visser W, Williams L (eds) Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016—Companion Volume. ACM, New York, pp 589–592. https://doi.org/10.1145/2889160.2889163
29. Müller P, Schwerhoff M, Summers AJ (2016) Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann B, Leino KRM (eds) Verification, model checking, and abstract interpretation—17th international conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings. Lecture notes in computer science. Springer, Heidelberg, vol 9583, pp 41–62. https://doi.org/10.1007/978-3-662-49122-5_2
30. Ho S, Protzenko J (2022) AENEAS: Rust verification by functional translation. Proc ACM Program Lang 6(ICFP):711–741. https://doi.org/10.1145/3547647
31. Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang Y (2002) Cyclone: a safe dialect of C. In: Ellis CS (ed) Proceedings of the general track: 2002 USENIX annual technical conference, June 10–15, 2002, Monterey, California, USA, pp. 275–288. USENIX, Berkeley (2002). http://www.usenix.org/publications/library/proceedings/usenix02/jim.html
32. Tofte M, Talpin J (1994) Implementation of the typed call-by-value lambda-calculus using a stack of regions. In: Boehm H, Lang B, Yellin DM (eds) conference record of POPL'94: 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages, Portland, Oregon, USA, January 17–21, 1994. ACM Press, New York, pp 188–201. https://doi.org/10.1145/174675.177855
33. Balabonski T, Pottier F, Protzenko J (2016) The design and formalization of Mezzo, a permission-based programming language. ACM Trans Program Lang Syst 38(4):14–11494
34. Tov JA, Pucella R (2011) Practical affine types. In: Ball T, Sagiv M (eds) Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, Austin, TX, USA, January 26–28, 2011. ACM, New York, pp 447–458. https://doi.org/10.1145/1926385.1926436
35. Boyland J, Noble J, Retert W (2001) Capabilities for sharing: a generalisation of uniqueness and read-only. In: Knudsen JL (ed) ECOOP 2001—object-oriented programming, 15th European conference, Budapest, Hungary, June 18–22, 2001, Proceedings. lecture notes in computer science Springer, Heidelberg, vol 2072, pp 2–27. https://doi.org/10.1007/3-540-45337-7_2
36. Clebsch S, Drossopoulou S, Blessing S, McNeil A (2015) Deny capabilities for safe, fast actors. In: Boix EG, Haller P, Ricci A, Varela C (eds) Proceedings of the 5th international workshop on programming based on actors, agents, and decentralized control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015. ACM, New York, pp 1–12. https://doi.org/10.1145/2824815.2824816
37. Stork S, Naden K, Sunshine J, Mohr M, Fonseca A, Marques P, Aldrich J (2014) Æminium: a permission based concurrent-by-default programming language approach. In: O'Boyle MFP, Pingali K (eds) ACM SIGPLAN conference on programming language design and implementation, PLDI'14, Edinburgh, United Kingdom—June 09–11, 2014. ACM, New York, p 26. https://doi.org/10.1145/2594291.2594344
38. Matsushita Y, Denis X, Jourdan J, Dreyer D (2022) Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In: Jhala R, Dillig I (eds) PLDI'22: 43rd ACM SIGPLAN

international conference on programming language design and implementation, San Diego, CA, USA, June 13–17, 2022. ACM, San Diego, pp 841–856. https://doi.org/10.1145/3519939.3523704
39. Weiss A, Patterson D, Matsakis ND, Ahmed A (2019) Oxide: the essence of Rust arXiv: 1903.00982
40. Wang F, Song F, Zhang M, Zhu X, Zhang J (2018) KRust: a formal executable semantics of Rust arXiv: 1804.10806