KRust: A Formal Executable Semantics of Rust

Feng Wang*, Fu Song*, Min Zhang†, Xiaoran Zhu† and Jun Zhang*
*School of Information Science and Technology, ShanghaiTech University, Shanghai, China
†Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China

***Abstract*—Rust is a new and promising high-level system programming language. It provides both memory safety and thread safety through its novel mechanisms such as ownership, moves and borrows. Ownership system ensures that at any point there is only one owner of any given resource. The ownership of a resource can be moved or borrowed according to the lifetimes. The ownership system establishes a clear lifetime for each value and hence Rust does not necessarily need garbage collection. These novel features bring Rust high performance, fine-grained low-level control over memory without garbage collection, which differentiate Rust from other existing prevalent languages. For formal analysis of Rust programs and helping programmers learn its new mechanisms and features, a formal semantics of Rust is desired and useful as a fundament for developing related tools. In this paper, we present a formal executable operational semantics of a subset of Rust, called KRust. The semantics is defined in $\mathbb{K}$, a rewriting-based executable semantic framework for programming languages. The executable semantics yields automatically a formal interpreter and verification tools for Rust programs. KRust has been validated by testing with 182 tests, including 157 tests from the official Rust test suite. We individually found an error in the Rust compiler.**

***Index Terms*—Formal operational semantics, Rust programming language, $\mathbb{K}$ framework**

I. Introduction

Recently, a new system programming language Rust was designed and implemented by Mozilla [1], aiming at achieving both high-level safety and low-level control in developing system software, such as operating systems, device drivers, game engines and web browsers. Like most modern high-level languages, Rust guarantees memory safety and thread safety, and meanwhile it supports zero-cost abstractions for many common system programming idioms and provides fine-grained low-level control over the use of memory, without needing a garbage collector. One key to meeting all these promises is Rust's novel system of ownership, moves and borrows. The ownership system establishes a clear lifetime for each value, making garbage collection unnecessary in the core language. Moreover, it also prevents data-races at compile-time. Ownership, moves and borrows are checked at compile time and are carefully designed to complement its static linear type system. Rust has been used to implement various types of systems such as operating systems [2], parallel browser engine [3] and Intel SGX Enclave [4] (for organizations using Rust for software development, see [5]).

This work was supported primarily by the National Natural Science Foundation of China (NSFC) grants 61532019 and 61761136011. Min Zhang was supported by the NSFC grant 61502171.

The new features make the semantics of Rust different from other common languages, which brings new difficulties in reasoning about Rust programs. The difficulty constantly baffles developers and has become a common topic of question-and-answer websites (for instance, [6]). The ownership of an object is a variable binding. When a variable goes out of its scope, Rust will free the bound resources. The ownership of an object can be *transferred*, after which the object cannot be accessed via the outdated owner. This ensures that there is exactly one binding to any object. Instead of transferring ownership, Rust provides borrows which allows an object to be shared by multiple references. There are two kinds of borrows in Rust: *mutable references* and *immutable references*. A mutable reference can mutate the borrowed object if the object is mutable, while an immutable reference cannot mutate the borrowed object even if the object itself is mutable. The basic ownership discipline enforced by Rust is that an object shared by multiple references is immutable. This property eliminates a wide range of common low-level programming errors, such as use-after-free, data races, and iterator invalidation. These specific semantic rules are unusual, and hence the semantics of other modern programming languages such as C/C++, Java and Javascript cannot be directly adapted to Rust.

To remedy this situation, a formal semantics of Rust is desired and useful as a fundament for reasoning about Rust programs in a formal way and developing related computer-aided tools. To our knowledge, the formal semantics of Rust has not yet been well studied, which impedes further developments of formal analysis and verification tools for Rust programs. In this paper, we make a major step toward rectifying this situation by giving the first formal operational semantics of a subset of Rust. We design a formal operational semantics of Rust capturing ownership, ownership moves and borrows. To avoid our semantics to be just "paper work", we formalize the semantics in the $\mathbb{K}$ framework [7] (http://kframework.org), a scalable semantic framework for programming languages which has been successfully applied to C [8] and Java [9]. We call the $\mathbb{K}$ definition of the semantics KRust. [1] To the best of our knowledge, it is the first formal executable semantics for Rust.

There are several benefits from the formal semantics defined in the $\mathbb{K}$ framework. Firstly, the semantics defined in $\mathbb{K}$ is both machine readable and executable, from which an interpreter of Rust is generated automatically. Being executable, the seman-

[1] KRust and all the related sample examples are available for downloading from: http://sist.shanghaitech.edu.cn/faculty/songfu/Projects/KRust.

tics has been tested with 182 tests, including 157 tests from the official Rust test suite. We individually found an error in the Rust compiler. Secondly, $\mathbb{K}$ provides a simple notation for modular semantics of languages, making the semantics easy to define and extensible. The semantics offers a formal reference and a correctness definition for implementers of tools such as parsers, compilers, interpreters and debuggers, which would greatly facilitate developers' understanding, freeing them from lengthy, ambiguous, elusive Rust documentations. Moreover, the semantics could automatically yield formal analysis tools such as state-space explorer for reachability, model-checker, symbolic execution engine, deductive program verifier by the language-independent tools provided in $\mathbb{K}$ [10].

Organization. Section II gives a brief overview of Rust. In Section III, after introducing the basic notations of $\mathbb{K}$, we present the formal semantics `KRust` of Rust in $\mathbb{K}$. Conformance testing and applications of `KRust` are described in Section IV. Section V discusses related work. Finally, we conclude the work with a discussion in Section VI.

II. A Tour of Rust

In this section, we give a brief overview of Rust. Rust is a C-like programming language which contains common constructs from modern programming languages such as *if/else* branches, *while/for* loops, *functions*, *compound data structures*, etc. We will mainly point out distinct features of Rust, compared with other well-known modern programming languages such as C/C++ and Java.

A. Mutability

Variables are declared using `let` statements. By default, a variable is *immutable*, which means that its value cannot be mutated. To declare a mutable variable, `mut` is required in the declaration statement.

```
1  fn main(){
2    let x=9;
3    x=10; // Error!
4    let mut y = 0;
5    let mut z: bool;
6  }
```

For instance, the above code declares an immutable variable `x` at Line 2 and a mutable variable `y` at Line 4 whose types are inferred at compile-time. The type can also be explicitly specified in the program like the mutable variable `z` at Line 5. The Rust compiler will issue an error at Line 3, as the immutable variable `x` is reassigned. This is different from other modern programming languages like C/C++ and Java. The semantic rules of Rust should take the mutability of variables into account.

B. Functions

Functions are declared with the keyword `fn` and each of them should return *exactly one value*. There are two ways to return a value if the function definition declares a return type. The first one is to return a value in the body of the function explicitly using the `return` statement. The other one is to return the value of the last expression in the body of the function, if there is no explicit return statement. However, if the function definition does not declare a return type, Rust will implicitly returns the unit type (). Indeed, a function definition without declaring a return type is just syntactic sugar for the same function definition with return type (). Furthermore, Rust has no restriction on the order of function definitions, namely that Rust programs can invoke functions which are defined later. These features are unusual and introduce tricky corner cases.

```
1  fn foo(x:i32, y:i32) -> i32 {
2    x+y // return x+y;
3  }
```

The above program defines a function `foo` which takes two 32-bit integers `x` and `y` as arguments and returns `x+y`. This function behaves the same as the function which replaces the last expression `x+y` in `foo` with the return statement `return x+y;`.

C. Ownership

Ownership is the key feature of Rust, which guarantees memory safety and thread safety without garbage collection. The basic form of ownership is *exclusive ownership*, namely that each object has a *unique* owner at any time. This ensures that at most one reference is allowed to mutate a given location. When an object is created and assigned to a variable `x`, the variable `x` becomes the owner of the object. If the object is reassigned (as well as parameter passing, etc.) to another variable `y`, the ownership of the object is *transferred* from the variable `x` to the variable `y`, namely that `y` becomes the owner of the object and `x` is not the owner of the object. This is so-called *move semantics* in Rust. This ownership discipline rules out aliasing entirely, and thus prevents data race. Moreover, if the owner of the object goes out of scope, the object is destroyed automatically without garbage collection. This is implemented by the Rust compiler which inserts a call to a destructor at the end of the owner's scope, called *drop* in Rust. This ownership discipline enforces automatic memory management and prevents other errors commonplace in low-level pointer-manipulating programs, like use-after-free or double-free. To see this principle in practice, consider the following sample program.

```
1  struct Point{ x: i32, y: i32, }
2  fn main(){
3    let p = Point {x:1, y:2};
4    {
5      let mut q = p; //q becomes the owner
6      q.x = 2;
7      println!("{}",p.x);  // Error!
8    }
9    println!("{}",q.x);  // Error!
10 }
```

`Point` is a compound data structure consisting of two 32-bit integer fields `x` and `y`. At Line 3, a `Point` object is created and assigned to the mutable variable `p`. The Rust compiler will issue an *error* at Line 7 which accesses the `x` field of the `Point` object via the reference `p`, as the ownership of the `Point` object was already transferred from `p` to `q` at Line 5. Moreover, the Rust compiler will also issue another *error* at Line 9 which accesses the `x` field of the `Point` object via the

reference q, as the owner q went out of its scope and the Point object was already destroyed. These scenarios rarely happen in existing prevalent programming languages, thus makes the Rust semantics unusual.

D. Borrowing

The ownership discipline is a fairly straightforward mechanism for guaranteeing memory safety and thread safety. However, it is also too restrictive to develop industry programs. To address this issue, Rust provides a mechanism used to handle references, called *borrowing*. There are two different borrowing (i.e., reference types) in Rust: *mutable references* and *immutable references*. A mutable reference grants temporary *exclusive read and write* access to the object, i.e., each object has at most one mutable reference (without any immutable references). This ensures that mutable references are always unique pointers. A mutable reference can be *reborrowed* to someone else. Contrary to mutable references, an immutable reference grants temporary *read-only* access to the object and it is allowed that multiple immutable references refer to the same object.

```
1  fn main() {
2      let x1 = 1; // or let mut x1 = 1;
3      let p1 = &x1; // x1 is borrowed immutably
4      let q1 = &x1; // x1 is borrowed immutably
5      *p1 = 2; // Error!
6      let y = &mut x1; // Error!
7
8      let mut x2 = 1;
9      let p2 = &mut x2; // x2 is borrowed mutably
10     x2 = 2; // Error!
11     *p2 = 2; // OK!
12     let p3 = &mut x2;  // Error!
13 }
```

We can see borrowing in action in the above example. The variable x1 is immutably borrowed twice at Lines 3 and 4. The compiler will issue an *error* at Line 5 which tries to mutate the value of x1. It also issues another *error* at Line 6, since the immutable variable x1 cannot be mutably borrowed. Line 9 shows that the mutable variable x2 can be mutably borrowed. Line 10 demonstrates that the value of x2 cannot be mutated via x2 once it is borrowed. Instead, it can be mutated via the mutable reference p2 at Line 11. Line 12 shows that the mutable variable x2 cannot be mutably borrowed more than once.

E. Lifetime

Borrowing grants *temporary* access to the object. Rust associates to each reference a *lifetime* to specify how long borrowing lasts. Intuitively, lifetimes are effectively just names for scopes somewhere in the program, but they are not the same. The lifetime of a reference should be included in the lifetime of the borrowed variable. Rust provides a convention so that lifetimes can be elided in general, which is why they did not show up in the above examples. Rust also supports named lifetimes which helps the Rust compiler to aggressively infer lifetimes and makes sure all borrows are valid.

```
1  fn main(){
2      let mut x ;
3      {
4          let y = 1;
5          x = &y;    // Error!
6      }
7      let z = 1;
8      let p = &z; // OK!
9  }
```

The above example illustrates intuition behind lifetime. There is an *error* at Line 5 as the lifetime of x is not included in the lifetime of y. Instead, it is fine to borrow z at Line 8, as the lifetime of p is included in the lifetime of z. Therefore, Rust's variable context is *substructural*.

III. KRust: The Formal Semantics of Rust in $\mathbb{K}$

In this section, we first introduce the basic notations of $\mathbb{K}$, and then define the addressed formal syntax of Rust. Finally, we define the configurations and formal semantics KRust of Rust in $\mathbb{K}$.

A. $\mathbb{K}$ Framework

$\mathbb{K}$ Framework is a rewrite-based executable semantic framework. Operational semantics of a programming language can be formally defined with the state of an executing program being represented as a configuration and the semantics of each program statement being defined as $\mathbb{K}$ rules. With the defined operational semantics, $\mathbb{K}$ automatically generates an interpreter which can *execute* programs of the language. Besides, $\mathbb{K}$ also provides formal analysis functionalities such as model checking, symbolic execution, and theorem proving [10]. A number of programming languages have been formalized using $\mathbb{K}$, such as C [8], Python [11], PHP [12] and Java [9].

In this section, we briefly introduce the mechanism of how to define operational semantics in $\mathbb{K}$ with an example. Basically, the syntax of a language is defined using BNF with semantic attributes, and the operational semantics is defined by a set of $\mathbb{K}$ rules which describe the effect of atomic program statements over $\mathbb{K}$ configurations. A $\mathbb{K}$ configuration is essentially a nested cell defined in XML style, which specifies a state of an executing program.

For better understanding of $\mathbb{K}$, let us consider the following semantic rule for lookup function in Rust, which is used to lookup the value of a name (e.g., variable) when a statement which contains that name is being executed.

$$\left\langle \frac{X}{V} \cdots \right\rangle_{\mathsf{k}} \langle \cdots X \mapsto L \cdots \rangle_{\mathsf{env}} \langle \cdots L \mapsto V \cdots \rangle_{\mathsf{store}}$$

There are three cells related to the lookup function. The cell k (i.e., the cell labeled with k) is used to store the computations of a program to be executed. In this example, X is the next expression/statement to evaluate/execute, where X is a $\mathbb{K}$ label representing a program name. Both env and store cells are Map types to store key-value pairs, where env is used to store the mapping from program names to locations in the form of $X \mapsto L$, and store is used to store the mapping from locations to values in the form of $L \mapsto V$. The dots "$\cdots$" in the cells are structural frames, denoting irrelevant pairs or computations.

For instance, given the following two statements:

```
1  let a = 1;
2  let b = a;// b = 1
```

TABLE I
THE SYNTAX OF KRust

Syntax	Description
Id ::= [a-zA-Z_][a-zA-Z0-9_]*	Identifier
Type ::= i8 | u8 | i16 | u16 | i32 | u32 | i64 | u64 | f32 | f64 | isize | usize | char | &str | bool | Id | () | [Type;Exp] | fn (Types) -> Type Types ::= Type*	Variable Types
TypedId ::= Id : Type TypedIds ::= TypedId*	Auxiliary types
ConstAndStatic ::= `const` Id : Type = Exp ; | `static mut` ? Id : Type = Exp ; DeclExp ::= `let mut` ? Id [: Type] ? [= Exp] ? | ConstAndStatic;	Variable declaration
Op ::= "+" | "-" | "*" | "/" | "%" | "|" | "&" | ">>" | "<<" | "<" | "<=" | ">" | ">=" | "==" | "!=" | "||" | "&&" Exp ::= Int | Bool | Float | String | Char | Id | *Id | [Exps] | [Exp;Exp] | vec![Exps] | (Exp) | Exp[Exp] | {Exp} | Ref Exp | Id { StructValues } | Exp(Exp) | -Exp | ! Exp | Exp Op Exp Exps ::= Exp*	Expressions
AssignOp ::= "=" | "+=" | "-=" | "*=" | "/=" AssignmentStmt ::= Id AssignOp Exp ; | Id[Exp] AssignOp Exp ; | *Id AssignOp Exp; | Id . Id AssignOp Exp ;	Assignment statement
If ::= `if` Exp `else` ? Block While ::= `while` Exp Block Loop ::= `loop` Block | If | While	If, while and loop statement
Block ::= { } | { Stmts } | { Stmts Exp }	Block
Ref ::= & | &mut	Two types of references
Struct ::= `struct` Id { TypedIds } StructValue ::= Id : Exp StructValues ::= StructValue* StructInstance ::= Id { StructValues }	Struct
For ::= `for` Id `in` Int..Int Block	For statement
Function ::= `fn` Id (TypedIds) [-> Type] ? Block	Function
Stmts ::= DeclExp | AssignStmt | Block | Exp ; | return ; | return Exp ; | Loop ; | Loop | Function | Struct | For	Statements

Variable a should be first evaluated to 1 using the lookup rule when the second statement is being executed.

B. Syntax

Table I presents the syntax of a subset of Rust defined in KRust. The syntax is described by a dialect of Extended Backus-Naur Form (EBNF) according to the grammar of Rust [13]. We use two repetition forms in the definition of the syntax, where "?" means zero or one repetition and "*" means zero or more repetitions. The option is represented through squared brackets [...] followed with "?". For instance, in syntax for function declaration, "[-> Type] ?" means that "-> Type" maybe present just once, or not at all. Since Rust shares many common conventions with prevalent functional and imperative programming languages, most of its syntax are easy to understand and hence we omit corresponding explanations. We remark that Table I is not a full syntax of Rust, e.g., traits and pattern matching are excluded (cf. Section VI).

C. Configuration

A $\mathbb{K}$ configuration is represented by nested multisets of labeled cells. Figure 1 shows the 13 main cells in a configuration for the representation of the state of KRust programs.

The cell T is the top one in $\mathbb{K}$ which contains all the cells. As aforementioned, the k cell contains the computations of a program. The env cell is the local environment, recording the map from variables to their locations. Inside the control cell, there is a fstack cell which encodes the stack frame. The fstack cell is a list, in which each element contains an

$$\left\langle \begin{array}{c} \langle K \rangle_{\mathsf{k}}\, \langle Map \rangle_{\mathsf{env}}\, \langle \langle List \rangle_{\mathsf{fstack}} \rangle_{\mathsf{control}}\, \langle Map \rangle_{\mathsf{genv}} \\ \langle Map \rangle_{\mathsf{typeEnv}}\, \langle Map \rangle_{\mathsf{store}}\, \langle Map \rangle_{\mathsf{mutType}}\, \langle 0 \rangle_{\mathsf{nextLoc}} \\ \langle Map \rangle_{\mathsf{borrow}}\, \langle Map \rangle_{\mathsf{ref}}\, \langle Map \rangle_{\mathsf{refType}}\, \langle Map \rangle_{\mathsf{moved}} \end{array} \right\rangle_{\mathsf{T}}$$

Fig. 1. The $\mathbb{K}$ configuration for the states of KRust programs

env cell, some computations and a return type. The genv cell represents global environment. The typeEnv cell records the type of a given variable's location. The values of all the defined variables are stored in the store cell.

Since a variable is either mutable or immutable in Rust, we add mutType cell to record whether the variable is mutable or immutable. When a new variable is declared, a new location that is an integer value is allocated from nextLoc cell. After that, the integer value in nextLoc is increased by one. The borrow cell keeps the record whether a variable is mutably or immutably borrowed if there exists an alive reference to it. The ref cell records reference relations. The refType cell contains the types of references, including immutable and mutable references. The moved cell is a map from locations of variables to $\{0, 1\}$, where 1 denotes that the variable has been moved, otherwise not.

D. Formalization of the semantics

In this section, we present the formal semantics KRust in $\mathbb{K}$, emphasizing on key features of Rust such as: mutability, borrows and ownership. Table II shows the partial semantic rules of KRust, which specify the semantics of Rust statements

TABLE II
THE PARTIAL SEMANTIC RULES OF KRust

Variable declaration and assignment

$$\left\langle \frac{\texttt{let } X : T;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{lll} \langle X \Rightarrow L \rangle_{\text{env}} & \langle L \Rightarrow \bot \rangle_{\text{store}} & \langle L \Rightarrow T \rangle_{\text{typeEnv}} \\ \langle L \Rightarrow 0 \rangle_{\text{mutType}} & \langle L \Rightarrow \bot \rangle_{\text{borrow}} & \langle L \Rightarrow \bot \rangle_{\text{ref}} \\ \langle L \Rightarrow \bot \rangle_{\text{refType}} & \langle L \Rightarrow 0 \rangle_{\text{moved}} & \langle L+1 \rangle_{\text{nextLoc}} \end{array} \right\}$$

[Declaration-of-Immutable-Variable]

$$\left\langle \frac{\texttt{let mut } X : T;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{lll} \langle X \Rightarrow L \rangle_{\text{env}} & \langle L \Rightarrow \bot \rangle_{\text{store}} & \langle L \Rightarrow T \rangle_{\text{typeEnv}} \\ \langle L \Rightarrow 1 \rangle_{\text{mutType}} & \langle L \Rightarrow \bot \rangle_{\text{borrow}} & \langle L \Rightarrow \bot \rangle_{\text{ref}} \\ \langle L \Rightarrow \bot \rangle_{\text{refType}} & \langle L \Rightarrow 0 \rangle_{\text{moved}} & \langle L+1 \rangle_{\text{nextLoc}} \end{array} \right\}$$

[Declaration-of-Mutable-Variable]

$$\left\langle \frac{X}{V} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L \rangle_{\text{env}} & \\ \langle L \mapsto V \rangle_{\text{store}} & \end{array} \right\}$$

[Lookup-of-Variable]

$$\left\langle \frac{X=V;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L \rangle_{\text{env}} & \langle L \mapsto T \rangle_{\text{typeEnv}} \\ \langle L \mapsto 1 \rangle_{\text{mutType}} & \langle L \mapsto (\_ \Rightarrow V) \rangle_{\text{store}} \end{array} \right\}$$

when $T = getType(V)$ [Assignment]

$$\left\langle \frac{\texttt{let mut } X\texttt{: } [T;N];}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \Rightarrow L \rangle_{\text{env}} & \langle L \Rightarrow [T;N] \rangle_{\text{typeEnv}} \\ \langle L \Rightarrow 1 \rangle_{\text{mutType}} & \langle L+N \rangle_{\text{nextLoc}} \\ \langle L \ldots L+N-1 \Rightarrow \bot \rangle_{\text{store}} & \end{array} \right\}$$

[Declaration-of-Mutable-Array]

$$\left\langle \frac{X[I] \texttt{ = } V;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L \rangle_{\text{env}} & \langle L \mapsto [T;N] \rangle_{\text{typeEnv}} \\ \langle L \mapsto 1 \rangle_{\text{mutType}} & \langle L+I \mapsto (\_ \Rightarrow V) \rangle_{\text{store}} \end{array} \right\}$$

when $0 \le I < N$ and $T = getType(V)$ [Updating-of-Array-Element]

$$\left\langle \frac{X[I]}{V} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L \rangle_{\text{env}} & \langle L \mapsto [T;N] \rangle_{\text{typeEnv}} \\ \langle L+I \mapsto V \rangle_{\text{store}} & \end{array} \right\}$$

when $0 \le I < N$ [Evaluation-of-Array-Element]

Borrow, reference, lifetime and dereference

$$\left\langle \frac{X \texttt{ = \&mut } Y;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L_1, Y \mapsto L_2 \rangle_{\text{env}} & \langle L_1 \mapsto 1, L_2 \mapsto 1 \rangle_{\text{mutType}} \\ \langle L_1 \mapsto 0, L_2 \mapsto 0 \rangle_{\text{moved}} & \langle L_1 \mapsto (\_ \Rightarrow L_2) \rangle_{\text{ref}} \\ \langle L_1 \mapsto (\_ \Rightarrow 1) \rangle_{\text{refType}} & \langle L_2 \mapsto (\bot \Rightarrow 1) \rangle_{\text{borrow}} \end{array} \right\}$$

when $L_1 > L_2$ [Mutable-Reference]

$$\left\langle \frac{X \texttt{ = \&} Y;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L_1, Y \mapsto L_2 \rangle_{\text{env}} & \langle L_1 \mapsto 1, L_2 \mapsto 1 \rangle_{\text{mutType}} \\ \langle L_1 \mapsto 0, L_2 \mapsto 0 \rangle_{\text{moved}} & \langle L_1 \mapsto (\_ \Rightarrow L_2) \rangle_{\text{ref}} \\ \langle L_1 \mapsto 0 \rangle_{\text{refType}} & \langle L_2 \mapsto 0 \rangle_{\text{borrow}} \end{array} \right\}$$

when $L_1 > L_2$ [Immutable-Reference]

$$\left\langle \frac{*X}{V} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L_1 \rangle_{\text{env}} & \langle L_1 \mapsto L_2 \rangle_{\text{ref}} \\ \langle L_1 \mapsto 0 \rangle_{\text{moved}} & \langle L_2 \mapsto V \rangle_{\text{store}} \end{array} \right\}$$

[Dereference]

Function definition and function call

$$\left\langle \frac{\texttt{fn } F(\texttt{TIDs}) \texttt{ ->T\{S\}}}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle F \Rightarrow L \rangle_{\text{env}} & \langle L+1 \rangle_{\text{nextLoc}} \\ \langle L \Rightarrow \lambda(\texttt{TIDs}, \texttt{S}, \texttt{T}) \rangle_{\text{store}} & \end{array} \right\}$$

[Function-Definition-with-Return-Type]

$$\left\langle \frac{\texttt{mkDecls((}X \texttt{ : } T, \texttt{ TIDs), (}V\texttt{,Vs))}}{\texttt{let } X \texttt{ : } T\texttt{=}V\texttt{; mkDecls(TIDs, Vs)}} \right\rangle_{\text{k}}$$

[mkDecls-Helper-Function]

$$\left\langle \frac{\lambda(\texttt{TIDs,S,T})(\texttt{Vs}) \rightsquigarrow K}{\texttt{mkDecls(TIDs,Vs)}\rightsquigarrow \texttt{S}\rightsquigarrow \texttt{return ();}} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle (\texttt{Env}, K, \texttt{T}) \rangle_{\text{fstack}} & \\ \langle \texttt{Env} \leftarrow \bot \rangle_{\text{env}} & \end{array} \right\}$$

[Function-Call]

$$\left\langle \frac{\texttt{fn } F(\texttt{TIDs}) \texttt{ \{S\}}}{\texttt{fn } F(\texttt{TIDs}) \texttt{ -> () \{S\}}} \right\rangle_{\text{k}}$$

[Function-Definition-without-Return-Type]

$$\left\langle \frac{\texttt{fn } F(\texttt{TIDs}) \texttt{ -> T \{S E\}}}{\texttt{fn } F(\texttt{TIDs}) \texttt{ -> T \{S return E;\}}} \right\rangle_{\text{k}}$$

[Function-Definition-Return-by-Last-Expression]

$$\left\langle \frac{\texttt{return } V\texttt{;}\rightsquigarrow -}{V \rightsquigarrow K} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle (\texttt{Env}, K, \texttt{T}) \rangle_{\text{fstack}} & \\ \langle \_ \leftarrow \texttt{Env} \rangle_{\text{env}} & \end{array} \right\}$$

when $T = getType(V)$ [Return]

Struct and ownership

$$\left\langle \frac{\texttt{struct } Z \texttt{ \{TIDs\}}}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle Z \Rightarrow L \rangle_{\text{env}} & \langle L+1 \rangle_{\text{nextLoc}} \\ \langle L \Rightarrow F_1(\texttt{TIDs}) \rangle_{\text{store}} & \end{array} \right\}$$

[Struct-Definition]

$$\left\langle \frac{\texttt{let mut } X \texttt{ = } Z \texttt{ \{Vs\};}}{F_2(F_1(\texttt{TIDs}),X,\texttt{Vs})} \right\rangle_{\text{k}} \left\{ \begin{array}{lll} \langle X \Rightarrow L_1,\ Z \mapsto L_2 \rangle_{\text{env}} & \langle L_1 \Rightarrow 1 \rangle_{\text{mutType}} & \langle L_1 \Rightarrow 0 \rangle_{\text{moved}} \\ \langle L_1 \Rightarrow Z \rangle_{\text{typeEnv}} & \langle L_2 \mapsto F_1(\texttt{TIDs}) \rangle_{\text{store}} & \langle L+1 \rangle_{\text{nextLoc}} \end{array} \right\}$$

[Declaration-of-Struct-Instance]

$$\left\langle \frac{F_2(F_1((Y \texttt{ : } T, \texttt{ TIDs}), X, (Y \texttt{ : } V), \texttt{Vs}))}{F_2(F_1(\texttt{TIDs}), X, \texttt{Vs})} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X.Y \Rightarrow L \rangle_{\text{env}} & \\ \langle L \Rightarrow V \rangle_{\text{store}} & \\ \langle L+1 \rangle_{\text{nextLoc}} & \end{array} \right\}$$

[F_1-F_2-Helper-Function]

$$\left\langle \frac{X \texttt{ = } Y;}{F_3(X,\ Y,\ \texttt{TIDs}) \rightsquigarrow F_4(Y)} \right\rangle_{\text{k}} \left\{ \begin{array}{lll} \langle X \mapsto L_1, Y \mapsto L_2, Z \mapsto L_3 \rangle_{\text{env}} & \langle L_1 \mapsto Z, L_2 \mapsto Z \rangle_{\text{typeEnv}} & \\ \langle L_1 \mapsto 1 \rangle_{\text{mutType}} & \langle L_3 \mapsto F_1(\texttt{TIDs}) \rangle_{\text{store}} & \end{array} \right\}$$

[Ownership-Move]

$$\left\langle \frac{F_3(X,Y,((P:T),\texttt{TIDs}))}{X.P \texttt{ = } Y.P\texttt{; } \rightsquigarrow \ F_3(X,Y,\texttt{TIDs})} \right\rangle_{\text{k}}$$

[F_3-Helper-Function]

$$\left\langle \frac{F_4(X)}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X \mapsto L \rangle_{\text{env}} & \\ \langle L \mapsto (0 \Rightarrow 1) \rangle_{\text{moved}} & \end{array} \right\}$$

[F_4-Helper-Function]

$$\left\langle \frac{X.Y}{V} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X.Y \mapsto L_1,\ X \mapsto L_2 \rangle_{\text{env}} & \\ \langle L_1 \mapsto V \rangle_{\text{store}} & \langle L_2 \mapsto 0 \rangle_{\text{moved}} \end{array} \right\}$$

[Evaluation-of-Struct-Field]

$$\left\langle \frac{X.Y \texttt{ = } V;}{\cdot} \right\rangle_{\text{k}} \left\{ \begin{array}{ll} \langle X.Y \mapsto L_1, X \mapsto L_2 \rangle_{\text{env}} & \langle L_2 \mapsto 1 \rangle_{\text{mutType}} \\ \langle L_1 \mapsto (\_ \Rightarrow V) \rangle_{\text{store}} & \langle L_2 \mapsto 0 \rangle_{\text{moved}} \end{array} \right\}$$

[Updating-of-Struct-Field]

by modifying the content of relevant cells. In the semantic rules, operator $\Rightarrow$ is used to add a new pair to its corresponding cell. For instance, $\langle X \Rightarrow L \rangle_{\text{env}}$ means inserting a new pair $X : L$ into the env cell. The operator $\bot$ denotes *undefined value*. Dot $\cdot$ is a special character in $\mathbb{K}$, representing the identity element in list and map structures. Note that curly braces are used only for the compactness of the rules, where we leave out all the dots "$\cdots$" in cells. The complete semantic rules of KRust can be found in its source code.

1) Variable declaration and assignment: The semantic rule [Declaration-of-Immutable-Variable] specifies how cells are affected after the execution of the statement `let` $X : T$; in the k cell, where X is a variable and T is a primitive type. A new pair $L : \bot$ is added to store, where $\bot$ indicates that X is uninitialized. The pair $L : 0$ is added to mutType, where 0 means that X is *immutable*. The next available location becomes $L + 1$ since X has consumed the location L. Some other initialization operations are made in the relevant cells as shown in the rule. The rule for the declaration of a mutable integer variable is defined likewise with a modification of the mutType cell.

The semantic rule [Assignment] is defined for the ordinary assignment, e.g., assigning a new value to an integer variable. It says that the assignment could be successfully executed only if X is mutable and the type of V is T, where *getType* is a predefined function to get the type of V. The execution updates the value in store using $L \mapsto (\_ \Rightarrow V)$, where $\_$ denotes that the current value of L can be any. The notation $A \mapsto (B \Rightarrow C)$ will be used regularly in this work, which stands for replacing the pair $A : B$ by $A : C$ in a cell. A declaration with an initial value can be handled similarly by combining the above semantic rules.

The semantic rule [Declaration-of-Mutable-Array] is defined for the declaration of a mutable array, where a new pair

$X : L$ is added to env, i.e., allocates a location L for X. In typeEnv, $[T; N]$ is the array type. $L \ldots L + N - 1$ in store denotes the locations $L, \cdots, L+N-1$ which are uninitialized. The integer in nextLoc cell is increased by N, as locations $L, \cdots, L+N-1$ are used to store the content of the array. The rules for the evaluation and updating of an array are defined likewise, as depicted in the table.

2) Borrowing, reference, lifetime and dereference: Both *immutable references* and *mutable references* for borrowing are implemented in `KRust`. [Mutable-Reference] and [Immutable-Reference] are two of semantic rules for immutable and mutable references respectively. The [Mutable-Reference] rule expresses that if both X and Y are *mutable* and have not been moved, and Y has not been borrowed yet, then the value of L_1 in ref cell is updated to L_2, which may be used for dereference, the reference type of L_1 in refType is assigned by 1 denoting that the reference type of L_1 is *mutable*, the pair $L_2 : \perp$ in cell borrow is updated to $L_2 : 1$ meaning that Y is now mutably borrowed, while the condition $L_1 > L_2$ ensures that X has to be declared later than Y, i.e., the lifetime of X is included in the lifetime of Y. The [Immutable-Reference] rule is defined likewise, except that Y is already immutably borrowed and the reference type of L_1 in refType is assigned by 0 denoting *immutable*. The intuition behind the [Dereference] rule is straightforward, namely, the value of $*X$ is evaluated via the reference relation $L_1 \mapsto L_2$ in the ref cell.

3) Function definition and function call: The standard function definition that has an explicit return type is handled by the [Function-Definition-with-Return-Type] rule, which allocates a location L for the name F in the env cell and binds the location L with an auxiliary function $\lambda()$ in store. The auxiliary function $\lambda()$ records the type and body of the function to define the semantics of function calls, i.e., the [Function-Call] rule.

The function call is processed into two steps: first replacing the function name by its $\lambda()$ function in store using the [Lookup-of-Variable] rule, then applying the [Function-Call] rule. The [Function-Call] rule first stores the current local environment `Env` together with the return type `T` and the remaining computations K into the fstack cell, reallocates a new empty local environment denoted by $\langle \texttt{Env} \leftarrow \perp \rangle_{\text{env}}$ in the rule, and sets `mkDecls(TIDs,Vs)` $\rightsquigarrow$ `S` $\rightsquigarrow$ `return ();` as the computations. The latter firsts executes the helper function `mkDecls()` which is used to declare formal parameters initialized with actual parameters `Vs` recursively (c.f. the [mkDecls-Helper-Function] rule), then executes the function body `S` followed by an additional return statement `return ();`. We remark that `return ();` is added to handle the case that the function definition does not have a return statement, and it will be executed only if there is no `return` statement at the end of the function body `S`.

The semantic rule [Function-Definition-without-Return-Type] handles the corner case that the function definition does not have a return type, for which, the unit type () is added as the return type. The semantic rule [Function-Definition-Return-by-Last-Expression] handles another corner case that the function definition returns the value of the last expression, for which, the last expression `E` is rewritten as a return statement `return E;`.

The rule [Return] first checks the type of the return value V, then returns the value V and finally restores the local environment `Env` and the computations K from the top of the fstack cell at the same time.

4) Struct and ownership: The rule [Struct-Definition] is used for struct definition, which adds a new pair $Z : L$ into the env cell. The value of L in store is a helper function F_1 that is used to record the fields of the struct.

The rule [Declaration-of-Struct-Instance] is defined for the declaration of a mutable struct instance. It allocates a location L_1 for X (i.e., adds $X : L_1$ into env) and sets the type of L_1 as Z, which is a struct name (i.e., add $L_1 : Z$ into typeEnv). Other related cells are initialized accordingly. The declaration and initialization of fields are handled by the computation $F_2(F_1(\texttt{TIDs}), X, \texttt{Vs})$, which is defined by the [F_1-F_2-Helper-Function] rule. This rule recursively allocates a location for each field, which is initialized with the corresponding value from `Vs`.

The rule [Ownership-Move] specifies Rust's move semantics, in which the assignment statement is encoded as the computations of two helper functions F_3 and F_4, if X and Y have same type and X is immutable. Notice that the pair $L_3 : F_1(\texttt{TIDs})$ in store ensures that Y is a struct instance. The semantics of the helper functions F_3 and F_4 are expressed by the rules [F_3-Helper-Function] and [F_4-Helper-Function] respectively. Intuitively, [F_3-Helper-Function] helps to copy the fields from Y to X. [F_4-Helper-Function] is used to update the move cell. We remark that $\langle L_1 \mapsto 0, L_2 \mapsto 0 \rangle_{\text{moved}}$ is not added in the [Ownership-Move] rule, as "$X.P = Y.P$;" in [F_3-Helper-Function] ensures that both variables are not moved yet. The semantic rules [Evaluation-of-Struct-Field] and [Updating-of-Struct-Field] are defined for evaluation and updating of a struct filed, in which it is required that the struct variable is not moved, i.e., the pair $L_2 : 0$ should occur in moved.

Remark that we incorporated the type and ownership checking of Rust into semantic rules. At first glance, all these checks are executed dynamically via semantic rules. We chose to define in this style instead of defining type rules and semantic rules individually based on the following two reasons. First, we believe it is more convenient for programmers to understand the semantics and type rules of Rust. Second, we could use the symbolic execution engine, turned from our semantics by the language-independent tools provided in $\mathbb{K}$ [10], to perform static type checking of Rust programs.

IV. Testing and Applications

In this section, we validate our semantics `KRust` and show some potential applications of `KRust`.

A. Conformance Testing

Following previous work [8], [12], [9], [14] which used test suite for validating executable language semantics, we tried to do the same. We validated our semantics `KRust` by testing

the KRust interpreter (that was automatically generated from the KRust semantics using the $\mathbb{K}$ framework) against both the official test suite of Rust [15] and hand-crafted tests.

The official test suite of Rust is used to test the Rust compiler. It is already split into folders containing different categories of tests. We chose tests from the "run-pass" folder, as others were designed for different purposes such as error message. There are 3119 tests in the "run-pass" folder: some of them can be compiled by the nightly version or stable version of the compiler, and some may be ignored during compiler testing, but it is unclear how these tests were used from the official documents. Therefore, we parsed all 3119 and 195 of them are supported by our syntax. In 195 tests, there are 38 tests that either do not have a *main* function or call some standard library functions. Therefore, we chose the remaining 157 tests.

Because 157 tests might not cover all the supported constructs, we hand-crafted 25 tests according to the syntax defined in Table I. 25 tests together cover all the primitive constructs.

We have tested the KRust interpreter against all these 182 tests. KRust successfully parsed all of them. The results produced by the interpreter are the same as the ones produced by the compiled programs using the Rust compiler, except the one that triggered an error in the Rust compiler. According to the informal semantics of Rust provided in the official documents, once the ownership of an object is transferred to another variable, the object cannot be accessed via the outdated owner. However, we found that fields of a struct object are still writable, but not readable, via the outdated owner. We have reported this issue to the Rust community [16]. We later found that a similar issue has already been reported in 2015 [17] and confirmed by the Rust community, but it has not been fixed yet.

Remark that the semantic coverage of the test suite has not been well-studied, we leave this to future work.

B. Applications

One of the main goals of our semantics is to provide a formal semantics for Rust. Beyond just giving a formal reference for the defined language, there are many applications of our formal semantics using the language-independent tools provided by $\mathbb{K}$ [10], such as state-space exploration for reachability, model-checking, symbolic execution and deductive program verification. In this work, we demonstrate this by showing two applications: *debugging* and *verification*, which are automatically derived from the semantics KRust. We omit demonstration of other applications in the paper since they have been well studied in other related works.

Debugging. We can turn the $\mathbb{K}$ debugger into a debugger for Rust which allows users to inspect program states. We demonstrate this by the following example.

```
1  fn main() {
2      let mut x: i32 = 10;
3      while x > 0 {
4          x = x - 1;
5      }
6  }
```

We can debug this program using the command.

```
krun test.rs --debugger
```

Users can step through one or more semantic rules individually from the current point and print the current state. For instance, after executing Line 4 once, part of the state will look like below:

```
<env> x |-> 1 </env>
<typeEnv> 1 |-> i32 </typeEnv>
<mutType> 1 |-> 1 </mutType>
<store> 1 |-> 9 </store>
```

Verification. A *sound-by-construction* program verifier for Rust can be automatically derived from KRust without additional effort. The verifier allows us to automatically check reachability properties including all Hoare-style functional correctness assertions and time complexity of a computation. As an example, we will verify the time complexity of the Euclidean algorithm by subtraction which computes greatest common divisor.

```
fn gcd(a: i32, b : i32) -> i32 {
    if a!=b {
        if a>b { return gcd(a-b, b); }
        else   { return gcd(a, b-a); }
    }else { return a; }
}
```

The Euclidean algorithm is implemented in Rust as shown above. We will prove its time complexity is indeed $\mathbf{O}(\max(a,b))$. To verify time complexity, an extra cell time is added to the configuration which increases a counter T each time when gcd is called. The core part of the specification for proving is:

```
gcd(X:Int, Y:Int)
<time> T1 => T2 </time>
requires X > 0 , Y > 0
ensures T2 - T1 <= maxInt(X,Y)
```

where T1 and T2 respectively denote the value of the counter T in pre-state and post-state of the function, requires and ensures respectively denote pre-condition and post-condition, T2 - T1 denotes the number of calls to gcd, and maxInt is a built-in function which returns the larger one of two integers. The verifier outputs true which proves that T2 - T1 <= maxInt(X,Y) holds, i.e., the time complexity is $\mathbf{O}(\max(a,b))$.

V. Related work

The semantics of various programming languages have been defined using the $\mathbb{K}$ framework, such as C11 [8], [18], Java 1.4 [9], JavaScript [14], PHP [12], Python 3.3 [11], Verilog [19], Scheme [20] and LLVM IR [21]. Compared to these works, the most distinguished aspect of our semantics is the formalization of the distinct features of Rust, namely *ownership*, *moves*, *borrows* and *lifetime*. To address these features, we had to redesign the *program state* instead of just copying from the existing works. Our semantics has been validated using the Rust standard test suite as well as hand-crafted tests.

In another research line, Rust has attracted some attention of researchers, and there are some works on the Rust language and Rust program verification.

Reed presented a formal semantics for Rust that captures the key features relevant to memory safety, unique pointers and borrowed references, described the operation of the Borrow Checker, and formalized the type system of Rust [22]. The main goal of this work is to provide a framework for borrow checking. However, Rust has been evolved a lot since this work and the semantics of this work has *not* been implemented yet, hence *not* executable. Very recently, Jung et al. defined a formal semantics and type system of a language λ_{Rust}, which incorporates Rust's notions [23]. Their work has been implemented in Coq. The main goal of this work is to study Rust's ownership discipline in the presence of unsafe code. However, the language λ_{Rust} is close to Rust's Mid-level Intermediate Representation (MIR) rather than *the actual Rust language*, and their semantics defines *more behaviors* than Rust does. Our semantics addresses the exact behavior of the actual Rust language and is executable.

Dewey et al. proposed a technique to fuzz type checker implementations and applied to test Rust's type checker [24]. It has identified 18 bugs. It is evident that formalization of the Rust type system is important, while formalization of the Rust semantics is the first step toward this.

Two Rust program analysis tools have been proposed [25], [26], which analyze Rust programs by translating them into the input languages of existing tools. But, transformation is prone to introduce inconsistencies and hence requires more effort to guarantee the correctness. Our semantics can be automatically turned into correct-by-construction formal analysis tools such as state-space explorer for reachability, model-checker and symbolic execution engine using the language-independent tools provided by $\mathbb{K}$ [10].

Independently, Kan et al. proposed an executable formal semantics of Rust in the $\mathbb{K}$ framework [27]. They defined a type system for a so-called *surface-Rust* language and a formal semantics on a core-language, instead of *the actual Rust language*. Hence, the interpreter and verification tools turned from their semantics using the language-independent tools provided by $\mathbb{K}$ are not *directly* applicable to Rust programs.

VI. Conclusion, Limitations and Future Work

In this work, we proposed a formal executable semantics `KRust` for Rust using the $\mathbb{K}$ framework. `KRust` captures (1) all the primitive types and their operations, (2) compound data types: *struct*, *array* and *vector*, (3) all the basic control flow constructs: *for*, *while*, *loop*, *if*, *if/else*, *function definition/call* and (4) three most distinct and important features: *ownership*, *borrow* and *lifetime*. We tested `Rust` on many hand-crafted tests and Rust's official tests suite. Tests in supported syntax are all passed. We also demonstrated potential applications of `KRust` for debugging and verification of Rust programs.

However, `KRust` does not cover the full features of Rust, as Rust is being actively developed and some of these features are not stable so far. As a witness, although the Rust's community provides some syntax in EBNF [13], it is still far away from complete. This makes the formalization of Rust much difficult, as mentioned in [23]. The following is a list of features that we haven't implemented yet but plan to implement: (1) structs with reference fields, (2) pattern matching which can be seen as a generalization of switch-case, (3) trait objects which like interfaces in Java, (4) lifetime annotation which is used to mark explicit lifetime in functions or structs, (5) complex closures which use outside variables, (6) concurrency for writing multi-threading programs, (7) crates and modules which are used to call external library codes (8) unsafe which is used to write code that the Rust compiler is unable to prove its safety, etc. A long-term program is to develop an almost complete formal executable semantics for Rust and formally verify Rust programs using formal analysis tools turned from the semantics, towards which the work reported in this paper is the first cornerstone.

References

[1] N. D. Matsakis and F. S. Klock II, "The Rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3, 2014, pp. 103–104.
[2] Redox, "Redox: a unix-like operating system written in Rust," https://www.redox-os.org, 2018.
[3] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *ICSE'16*, 2016, pp. 81–89.
[4] Y. Ding, R. Duan, L. Li, Y. Cheng, Y. Zhang, T. Chen, T. Wei, and H. Wang, "POSTER: Rust SGX SDK: towards memory safety in intel SGX enclave," in *CCS'17*, 2017, pp. 2491–2493.
[5] https://www.rust-lang.org/en-US/friends.html.
[6] Stackoverflow, "https://stackoverflow.com/search?q=rust+," 2018.
[7] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.
[8] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *POPL'12*, 2012, pp. 533–544.
[9] D. Bogdanas and G. Roşu, "K-Java: a complete semantics of Java," in *POPL'15*, 2015, pp. 445–456.
[10] G. Rosu, "$\mathbb{K}$: A semantic framework for programming languages and formal analysis tools," in *Dependable Software Systems Engineering*, 2017, pp. 186–206.
[11] D. Guth, "A formal semantics of Python 3.3," Master's thesis, University of Illinois at Urbana-Champaign, 2013.
[12] D. Filaretti and S. Maffeis, "An executable formal semantics of PHP," in *ECOOP'14*, 2014, pp. 567–592.
[13] https://doc.rust-lang.org/grammar.html.
[14] D. Park, A. Stefanescu, and G. Rosu, "KJS: a complete formal semantics of JavaScript," in *PLDI'15*, 2015, pp. 346–356.
[15] https://github.com/rust-lang/rust/tree/master/src/test.
[16] https://users.rust-lang.org/t/about-ownership/17120.
[17] https://github.com/rust-lang/rust/issues/21232.
[18] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of C," in *PLDI'15*, 2015, pp. 336–345.
[19] P. O. Meredith, M. Katelman, J. Meseguer, and G. Rosu, "A formal executable semantics of Verilog," in *MEMOCODE'10*, 2010, pp. 179–188.
[20] P. Meredith, M. Hills, and G. Rosu, "A K definition of Scheme," University of Illinois at Urbana-Champaign, Tech. Rep., 2007.
[21] LLVM IR in K, "http://github.com/davidlazar/llvm-semantics."
[22] E. Reed, "Patina: A formalization of the Rust programming language," University of Washington, Tech. Rep., 2015.
[23] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: securing the foundations of the Rust programming language," *PACMPL*, vol. 2, no. POPL, pp. 66:1–66:34, 2018.
[24] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust typechecker using CLP (T)," in *ASE'15*, 2015, pp. 482–493.
[25] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A bounded verifier for Rust," in *ASE'15*, 2015, pp. 75–80.
[26] F. Hahn, "Rust2viper: Building a static verifier for Rust," Master's thesis, ETH Zürich, 2016.
[27] S. Kan, D. Sanán, S. Lin, and Y. Liu, "K-Rust: An executable formal semantics for Rust," *CoRR*, vol. abs/1804.07608, 2018.