On the Termination of Borrow Checking in Featherweight Rust*

Étienne Payet[1], David J. Pearce[2], and Fausto Spoto[3]

[1] LIM, Université de La Réunion, France
[2] Victoria University of Wellington, New Zealand
[3] Dipartimento di Informatica, Università di Verona, Italy
etienne.payet@univ-reunion.fr
david.pearce@ecs.vuw.ac.nz
fausto.spoto@univr.it

Abstract. A distinguished feature of the Rust programming language is its ability to deallocate dynamically-allocated data structures as soon as they go out of scope, without relying on a garbage collector. At the same time, Rust lets programmers create references, called *borrows*, to data structures. A static borrow checker enforces that borrows can only be used in a controlled way, so that automatic deallocation does not introduce dangling references. Featherweight Rust provides a formalisation for a subset of Rust where borrow checking is encoded using flow typing [25]. However, we have identified a source of non-termination within the calculus which arises when typing environments contain cycles between variables. In fact, it turns out that well-typed programs cannot lead to such environments — but this was not immediately obvious from the presentation. This paper defines a simplification of Featherweight Rust, more amenable to formal proofs. Then it develops a sufficient condition that forbids cycles and, hence, guarantees termination. Furthermore, it proves that this condition is, in fact, maintained by Featherweight Rust for well-typed programs.

Keywords: Borrowing, Type Checking, Rust, Termination.

1 Introduction

The Rust programming language is seeing widespread use in areas such as system programming [1,5,15], blockchain systems [9,20], smart contracts [2,35] and more [6,3]. A key feature of Rust is its ability to automatically deallocate dynamically allocated data when it goes out of scope. This differs from most other programming languages, that either: require programmers to free data structures explicitly (e.g. C/C++); or, rely on garbage collection to free unreachable data (e.g. Java, C#, etc). The former approach is error prone (e.g. use-after-free

* Work supported by the SafePKT subproject of the LEDGER MVP Building Programme of the European Commission. Goal of the project is the analysis of Rust code used in the PKT blockchain (https://pkt.cash).

or free-after-free errors), whilst the latter is safe but costly (garbage collection consumes resources and data may not be released in a timely fashion).

In Rust, each data structure is *owned* by a variable [28]. Once that variable goes out of scope, the data is freed as well. Rust also allows data to be lent temporarily (e.g. as a function parameter) using *borrows*, which can be seen as pointers in traditional programming languages (but without ownership). Since borrows are access paths into data structures, the type checker of Rust must enforce strict rules on their creation and lifetime. For example, a location cannot be mutated as long as a borrow to it exists. To support this, data is divided into two categories: that which can be *copied* (e.g. primitives); and that which must be *moved* (e.g. mutable borrows). For the latter, assignments result in a transfer of ownership from rightvalue to leftvalue. The Rust compiler performs *borrow checking* to statically check that borrows are used safely (*ie.* that automatic deallocation does not create dangling pointers, that multithreaded code does not generate race conditions, etc).

Featherweight Rust (`FR`) formalises a subset of Rust and includes a proof of correctness for borrow checking [25]. In particular, borrow checking is formalised as a flow-sensitive type system, whose types include primitives (such as `int`), dynamically allocated data structures (collectively represented by a boxing operator) and borrows of leftvalues, both for *reading* (immutable borrows) and *writing* (mutable borrows). The type system rules are given by structural induction on the syntax of the Rust source code, and are hence well-founded. However, they use, internally, a procedure to type leftvalues. Since borrows include other leftvalues, we have discovered this procedure may enter an infinite loop and, in such case, the borrow checker would not terminate.

Contribution. This paper provides a sufficient condition which ensures that the borrow checker for Featherweight Rust terminates [25]. Our insight is that, for well-typed programs, this condition already holds for typing environments created during borrow checking. Hence, this is not a bug in Featherweight Rust *per se*, but rather an important condition which was left implicit. Our approach shows that data structures are *linearizable* at run time and, hence, that our condition holds for the specific kind of type environments the borrow checker builds during execution. This result is important in order to increase confidence in the borrow checker of Rust. Moreover, it provides a notion of well-foundness for the recursion used in the borrow checker, that future work can exploit in order to prove other properties by induction. For example, this is a necessary step towards a mechanical proof of Featherweight Rust.

2 Overview

This section illustrates various aspects of Rust related to memory allocation and borrowing, and provides an initial connection with Featherweight Rust (FR). A more detailed introduction to Rust can be found elsewhere [28,29].

Rust deallocates the data owned by a variable as soon as that variable goes out of scope. Consider the following, where the `Box::new(13)` allocates a new box on the heap which wraps the integer 13:

```
fn deallocate1() -> i32 {    // accepted by the borrow checker
    let x = Box::new(13);
    return 17;
}
```

Local variable `x` goes out of scope at the end of the function, hence Rust deallocates the box there, automatically. Assignments move the ownership of a value to their leftvalue. Consider the following:

```
fn deallocate2() {          // rejected by the borrow checker
    let x = Box::new(13);
    {
        let y = x;
    }
    println!("{}", x);
}
```

The assignment moves ownership of the box from `x` to `y`. Since `y` goes out of scope when the inner block ends, the box is deallocated there. Consequently, the print statement is trying to use deallocated data, *ie.* it is trying to access a dangling pointer. Correctly, the borrow checker of Rust rejects this. Consider the following function now:

```
fn ok1() -> Box<i32> {      // accepted by the borrow checker
    let x = Box::new(13);
    return x;
}
```

Here, ownership of the box is transferred from `x` to the return value, and subsequently to the caller of the function. When variable `x` reaches the end of its scope it no longer owns a value and, hence, Rust does not deallocate anything inside `ok1`.

Things become more complicated if borrows of data structures exist. For instance, the following function tries to return a borrow of a data structure that has been already deallocated:

```
fn dangling() -> &Box<i32> { // rejected by the borrow checker
    let i = Box::new(13);
    let result = &i;
    return result;
}
```

Local variable `i` owns the box and, when it goes out of scope at the end of the function, the box is deallocated. Variable `result` takes an *immutable borrow* of `i` (roughly a pointer to `i` without ownership). Thus, when the box is deallocated, `result` becomes a dangling pointer which cannot safely be returned. Again,

Rust rejects this function. Roughly, the borrow checker for FR [25] computes the following *typing* (or *type environment*) at the end of the function:

$$\{i \to \square\,\mathsf{int}, \texttt{result} \to \&i\}$$

For simplicity, FR uses `int` to collectively represent integer types in Rust (e.g. `i32`, `i64`, etc). Likewise, `□T` corresponds with `Box<T>` and provides the only form of dynamically allocated data in FR. Finally, $\&w$ (resp. `&mut` w) where w is a leftvalue is the type of an immutable (resp. mutable) borrow. Furthermore, since the borrow checker allows arbitrary leftvalues here (*ie.* not just variables), we can have types such as $\&{*}{*}y$.

Borrows are a sort of temporary ownership of a value. As a consequence, that value can be modified only through the borrow, for the whole duration of the borrow. Any other attempt to modify the value is rejected. Consider for instance the following function:

```
fn writes_to_borrowed() { // rejected by the borrow checker
    let v = 13;
    let w = 17;
    let mut y = &v;
    let x = &y;
    y = &w;
    println!("{}{}{}{}", x, y, v, w);
}
```

Here, the `y=&w` statement is trying to modify the leftvalue y that, however, has been borrowed at the previous line. Correctly, the borrow checker rejects this function. It computes the following typing just before the `y=&w` statement:

$$\{v \to \mathsf{int}, w \to \mathsf{int}, y \to \&v, x \to \&y\}$$

from where it is apparent that y is borrowed and, therefore, the subsequent assignment `y=&w` is rejected.

Borrows in previous examples are immutable: the borrowed value can be read from them, but cannot be modified from them. Borrows can also be mutable, meaning that they allow one to modify the borrowed value, with the dereference operator `*`. In this sense, a mutable borrow takes full responsibility about the borrowed value, for its whole lifetime. When a mutable borrow to a value exists, that value cannot be written *nor read* from any other path. Consider for instance the following function:

```
fn reads_mutably_borrowed() { // rejected by the borrow checker
    let mut z = 13;
    let y = &mut z;
    let x = z;
    println!("{}{}{}", x, y, z);
}
```

The statement `x=z` tries to read z, that has been mutably borrowed at the previous line. Hence, the borrow checker rejects this function. It computes the typing

$$\{z \to \mathsf{int}, y \to \&\texttt{mut}\ z\}$$

just before `x=z`, from where it is apparent that z is mutably borrowed there.

3 Preliminaries

This section gives a formal, simplified presentation of Featherweight Rust (`FR`) [25]. This retains the key features of `FR` relevant to our discussion but, for brevity, omits other aspects. Roughly speaking, the main simplifications are:

- **Compatibility**. The original formulation of `FR` supports a notion of *partial type*. This allows the "shadow" of a variable's type to be retained in the environment after it has been moved, such that subsequent re-assignments can be checked for compatibility. Since this is not important here, we reduce these shadow types to a single "dangling" type.
- **Borrows**. The original formulation of `FR` models borrows using *sets* of leftvalues. This allows `FR` to be easily extended with control-flow constructs, but is not strictly necessary for the core calculus. Since this makes our presentation more complex without adding anything significant, we restrict borrows to a single leftvalue.
- **Misc.** We have transformed some definitions, originally given as typing rules, into functions (such as type and move later). This makes them more compact and simplifies proofs involving them.

Definition 1 (LVals). *We assume a set of variables* Vars. *A context* $\kappa \subseteq \mathsf{Vars}$ *is a finite set of variables in scope. The set* LV_κ *of* leftvalues *over* κ *is:*

$$\mathsf{w} ::= x \mid *\mathsf{w}, \textit{where } x \in \kappa.$$

The root *of a leftvalue is then defined as:*

$$\begin{aligned} \mathsf{root}(x) &= x \qquad \textit{if } x \in \mathsf{Vars} \\ \mathsf{root}(*\mathsf{w}) &= \mathsf{root}(\mathsf{w}). \end{aligned}$$

Definition 2 (Expressions). *The set of* expressions e *is defined as*

$$\mathsf{e} ::= i \mid \mathsf{w} \mid \&\mathsf{w} \mid \&\texttt{mut}\ \mathsf{w} \mid \texttt{box}\ \mathsf{e}$$

Definition 3 (Terms). *We assume a set* Lifetimes *of* lifetimes l *which decorate blocks of code. The set of* terms t *is defined as (where* $x \in \mathsf{Vars}$ *and* $l \in \mathsf{Lifetimes}$*):*

$$\mathsf{t} ::= \ \mathsf{w} = \mathsf{e} \mid \texttt{let mut}\ x = \mathsf{e} \mid \{\ \mathsf{t}_1\ ;\ \ldots\ ;\ \mathsf{t}_n\ \}^l$$

Intuitively, variables declared in a block with lifetime l have lifetime l and are deallocated at the end of the block. Lifetimes are important for the borrow checker to ensure borrows do not outlive their referents and become dangling. The following illustrates a simple (invalid) program:

$$\{\ \texttt{let mut x} = 0;\ \texttt{let mut p} = \&\texttt{x};\ \{\ \texttt{let mut y} = 1;\ \texttt{p} = \&\texttt{y};\ \}^{\texttt{m}}\ \}^{\texttt{l}}$$

This program creates a dangling reference when the inner block completes and, hence, is rejected by the borrow checker.

The types used in FR are a simplification of those found in Rust, and include only primitive types (such as int) or structures dynamically allocated in memory (collectively represented by a box), but can also refer to a borrow or mutable borrow of a leftvalue.

Definition 4 (Types). *The set of* types *over a context κ is defined as follows (where $\mathsf{w} \in \mathsf{LV}_\kappa$):*

$$\mathsf{T}_\kappa ::= \ \mathsf{int} \mid \&\mathsf{w} \mid \&\texttt{mut}\ \mathsf{w} \mid\ \square\, \mathsf{T}_\kappa \mid \mathsf{dangling}$$

Here, type $\mathsf{dangling}$ is given to a variable whose value has been *moved*, that is, assigned to another owner.[4] Consequently, the value exists but cannot be accessed from that variable anymore.

Definition 5 (Declared Types). *The set of* declared types*, T^l, over κ associates types with lifetimes. We define $|T^l| = T$ and $\mathsf{lifetime}(T^l) = l$.*

Rust distinguishes types with *copy semantics* and types with *move semantics*. Values whose type has copy semantics are copied upon reading, while values whose type has move semantics are *moved* instead, in the sense that their original container loses the ownership to the value. Only mutable borrows and dynamically allocated data (*ie.* boxes) have move semantics.

Definition 6 (Copy and Move). *Let $T \in \mathsf{T}_\kappa$. Then T* has move semantics*, and we write $\mathsf{move}(T)$, if and only if $T = \&\texttt{mut}\ \mathsf{w}$ or $T = \square\, T'$ for some T'. In all other cases, T* has copy semantics*, and we write $\mathsf{copy}(T)$.*

Another useful notion is that of *full* types. They are types that do not contain $\mathsf{dangling}$. This notion is important because, as we will see in Sec. 4, only values with full type can be borrowed in Rust.

Definition 7 (Full type). *A type $T \in \mathsf{T}_\kappa$ is* full *if and only if* $\mathsf{dangling}$ *does not occur inside T. We write it as $\mathsf{full}(T)$.*

We define now the typings, or type environments, that is, information about the types of the variables in scope at a given program point, with their lifetime.

[4] This is a simplification of the $\mathsf{dangling}(T)$ type in [25], that embeds the *shadow* type T of a value that has been moved away.

Definition 8 (Typing). *Given a context κ, a* typing τ *over κ is a map from each variable $v \in \kappa$ to a type T and a lifetime l. We write this as $\tau(v) = T^l$.*

The types used in a typing can include borrows and mutable borrows. The basic idea of the borrow checker is that the root of the borrowed leftvalues (mutable or not) can only be used in a restricted way [25].

Definition 9 (Read/Write Prohibited). *Let κ be a context and τ a* typing *over κ. Then $\mathsf{w} \in \mathsf{LV}_\kappa$ is* read prohibited in τ*, written as* $\mathsf{readProhibited}(\mathsf{w}, \tau)$*, if* $\mathsf{root}(\mathsf{w})$ *occurs in a mutable borrow inside τ. Moreover,* w *is* write prohibited in τ*, written as* $\mathsf{writeProhibited}(\mathsf{w}, \tau)$*, if* $\mathsf{root}(\mathsf{w})$ *occurs in a borrow or in a mutable borrow inside τ.*

A typing provides type and lifetime information for variables in scope, and this naturally extends to leftvalues. The following is a translation[5] of Def. 3.11 in [25]. It can be seen as a recursive algorithm for typing leftvalues and, as such, it is heavily used in the borrow checker. The algorithm queries the typing when the leftvalue is actually a variable, and dereferences borrows and boxes when the leftvalue contains one or more $*$ operations, further recurring in the case of borrows. Types int and $\mathsf{dangling}$ cannot be dereferenced, hence the algorithm fails on them.

Definition 10 (LVal Typing). *Given a context κ, a typing τ over κ and $\mathsf{w} \in \mathsf{LV}_\kappa$, the partial function* $\mathsf{type}(\mathsf{w}, \tau)$ *yields the type and lifetime of* w *in τ:*

$$\mathsf{type}(x, \tau) = \tau(x)$$

$$\mathsf{type}(*\mathsf{w}, \tau) = \begin{cases} \textit{undefined} & \textit{if } \mathsf{type}(\mathsf{w}, \tau) \textit{ is undefined} \\ \textit{undefined} & \textit{if } |\mathsf{type}(\mathsf{w}, \tau)| = \mathsf{dangling} \\ \textit{undefined} & \textit{if } |\mathsf{type}(\mathsf{w}, \tau)| = \mathsf{int} \\ \mathsf{type}(\mathsf{w}', \tau) & \textit{if } |\mathsf{type}(\mathsf{w}, \tau)| = \&\mathsf{w}' \\ \mathsf{type}(\mathsf{w}', \tau) & \textit{if } |\mathsf{type}(\mathsf{w}, \tau)| = \&\texttt{mut } \mathsf{w}' \\ T^l & \textit{if } \mathsf{type}(\mathsf{w}, \tau) = (\square\, T)^l. \end{cases}$$

Def. 10 is clearly recursive, both on the structure of w and on the leftvalues contained in the borrows or mutable borrows that occur in the typing. In general, that recursion is not well-founded. In algorithmic terms, this means that this algorithm for typing leftvalues might not terminate. Consider for instance the typing $\{x \to \&*x\}$: the definition of $\mathsf{type}(*x, \tau)$ ends in an infinite loop. This example can be arbitrarily complicated, through the use of more involved cycles that pass through more variables. As a consequence, the natural question is to understand when the recursion in Def. 10 is well-founded and if that is always the case when it is used by the borrow checker of Featherweight Rust.

[5] This definition is given as a type system in [25] and as a recursive function here.

4 Borrow Checking

The borrow checker is formalized as a *flow-sensitive* type system [23] whose rules bind the typing τ *before* the evaluation of a term t to the typing τ' *after* that evaluation. We write this as $\tau, l \vdash \mathsf{t} \dashv \tau'$, where l is the *enclosing* lifetime of t (*ie.* that of the enclosing block). On expressions, the typing rules provide the inferred type T of the expression as well: $\tau, l \vdash e : T \dashv \tau'$.

4.1 Typing Expressions

T-Const. This rule applies to integer constants. Their evaluation yields a value of type int and does not modify the typing:

$$\frac{}{\tau, l \vdash i : \mathsf{int} \dashv \tau}$$

T-Copy. This rule applies to leftvalues whose type has copy semantics. Their evaluation yields their value, while the typing remains unchanged. The rule requires that the leftvalue can be accessed for reading:

$$\frac{T^m = \mathsf{type}(\mathsf{w}, \tau) \quad \mathsf{copy}(T) \quad \neg\mathsf{readProhibited}(\mathsf{w}, \tau)}{\tau, l \vdash \mathsf{w} : T \dashv \tau}$$

T-Move. This rule applies to leftvalues whose type has move semantics. Their evaluation yields their value, but the ownership of the value is moved away from the leftvalue. Because of this, the typing gets modified, by letting the old container of the value get the $\mathsf{dangling}$ type (i.e. so it cannot be used anymore). As a consequence, reading, from a leftvalue, a value with move semantics amounts to writing into its old container and requires write permission:

$$\frac{T^m = \mathsf{type}(\mathsf{w}, \tau) \quad \mathsf{move}(T) \quad \neg\mathsf{writeProhibited}(\mathsf{w}, \tau)}{\tau, l \vdash \mathsf{w} : T \dashv \mathsf{move}(\mathsf{w}, \tau)}$$

where the move function modifies the binding for the root of w:

$$\mathsf{move}(\mathsf{w}, \tau) = \tau[\mathsf{root}(\mathsf{w}) \mapsto \mathsf{strike}(\mathsf{w}, \tau(\mathsf{root}(\mathsf{w})))]$$

with

$$\mathsf{strike}(x, T^l) = \mathsf{dangling}^l$$
$$\mathsf{strike}(*\mathsf{w}, (\square\, T)^l) = (\square\, |\mathsf{strike}(\mathsf{w}, T^l)|)^l.$$

The function strike is undefined otherwise. We note also there are no cases for borrows since one cannot move out of a borrow in Rust.
T-ImmBorrow. The evaluation of a borrow expression requires the borrowed leftvalue to be readable and have full type (only values with full type can be borrowed in Rust):

$$\frac{\mathsf{full}(|\mathsf{type}(\mathsf{w}, \tau)|) \quad \neg\mathsf{readProhibited}(\mathsf{w}, \tau)}{\tau, l \vdash \&\mathsf{w} : \&\mathsf{w} \dashv \tau}$$

T-MutBorrow. The evaluation of a mutable borrow expression requires the borrowed leftvalue to be writable and have full type (only values with full type can be borrowed in Rust). Moreover, Rust requires that the borrowed leftvalue never traverses an immutable borrow:

$$\frac{\mathsf{full}(|\mathsf{type}(\mathsf{w},\tau)|) \quad \neg\mathsf{writeProhibited}(\mathsf{w},\tau) \quad \mathsf{mutable}(\mathsf{w},|\tau(\mathsf{root}(\mathsf{w}))|,\tau)}{\tau, l \vdash \&\texttt{mut}\ \mathsf{w} : \&\texttt{mut}\ \mathsf{w} \dashv \tau}$$

where

$$\begin{aligned}\mathsf{mutable}(x,T,\tau) &= \mathit{true}\\ \mathsf{mutable}(*\mathsf{w},\square\,T,\tau) &= \mathsf{mutable}(\mathsf{w},T,\tau)\\ \mathsf{mutable}(\underbrace{*\cdots*}_{n}x,\&\texttt{mut}\ \mathsf{w},\tau) &= \mathsf{mutable}(\underbrace{*\cdots*}_{n}\mathsf{w},|\tau(\mathsf{root}(\mathsf{w}))|,\tau).\end{aligned}$$

T-Box. The evaluation of a box expression simply recurs on the boxed expression:

$$\frac{\tau, l \vdash \mathsf{e} : T \dashv \tau'}{\tau, l \vdash \texttt{box}\ \mathsf{e} : \square\,T \dashv \tau'}$$

4.2 Typing Terms

T-Block. The execution of a block of statements simply recurs on each statement. At the end, the variables declared inside the block get dropped away. We assume that variables cannot be redefined inside a block, hence there is no risk of a name clash.

$$\frac{\tau, l \vdash \mathsf{t}_1 \dashv \tau_1 \quad \ldots \quad \tau_{n-1}, l \vdash \mathsf{t}_n \dashv \tau'}{\tau, l \vdash \{\mathsf{t}_1; \ldots; \mathsf{t}_n\}^m \dashv \mathsf{drop}(m,\tau')}$$

where

$$\mathsf{drop}(m,\tau) = \{x \to T^l \mid x \in \mathsf{dom}(\tau),\ \tau(x) = T^l \text{ and } l \neq m\}.$$

T-Declare. The declaration of a fresh variable x evaluates its initialization expression e and binds x to the type of e, decorated with the lifetime of the block of code where the declaration is evaluated:

$$\frac{x \notin \mathsf{dom}(\tau) \quad \tau, l \vdash \mathsf{e} : T \dashv \tau'}{\tau, l \vdash \texttt{let mut}\ x = \mathsf{e} \dashv \tau'[x \to T^l]}$$

T-Assign. The assignment of a value to a leftvalue w requires w to be writable. In that case, the assigned expression is evaluated and assigned to w. This is modelled through the write function below. Since w can be more complex than a single variable, the assignment might actually update a variable in a mutable borrow reachable from the root of w. This is reflected in the (quite complex) definition of write, that we take from [25] where more details can be found:

$$\frac{\begin{array}{c}\tau, l \vdash \mathsf{e} : T \dashv \tau' \quad \tau'' = \mathsf{write}(\tau',\mathsf{w},T) \quad \neg\mathsf{writeProhibited}(\mathsf{w},\tau'')\\ \mathsf{survives}(T,\mathsf{lifetime}(\mathsf{type}(\mathsf{w},\tau)),\tau')\end{array}}{\tau, l \vdash \mathsf{w} = \mathsf{e} \dashv \tau''}$$

where

$$\mathsf{write}(\tau, \underbrace{*\cdots*}_{n} x, T) = \mathsf{apply}(x, \mathsf{update}(\tau, n, |\tau(x)|, T))$$

where

$$\begin{aligned}
\mathsf{update}(\tau, 0, T', T) &= \langle \tau, T \rangle \\
\mathsf{update}(\tau, n+1, \Box\, T', T) &= \mathsf{expand}(\mathsf{update}(\tau, n, T', T)) \\
\mathsf{update}(\tau, n+1, \&\mathtt{mut}\ \mathsf{w}, T) &= \langle \mathsf{write}(\tau, \underbrace{*\cdots*}_{n} \mathsf{w}, T), \&\mathtt{mut}\ \mathsf{w} \rangle
\end{aligned}$$

and

$$\begin{aligned}
\mathsf{apply}(y, \langle \tau, T \rangle) &= \tau[y \to T^l] \quad \text{where } \mathsf{lifetime}(\tau(y)) = l \\
\mathsf{expand}(\langle \tau, T \rangle) &= \langle \tau, \Box\, T \rangle.
\end{aligned}$$

It is important to observe that if write modifies a type, it is that of x or that of variables inside the mutable borrows in τ.

Function $\mathsf{survives}(T, m, \tau)$ determines if all leftvalues contained in the borrows or mutable borrows inside the type T have a type whose lifetime is m or is larger than m. Hence they *survive* to the end of the lifetime m. The motivation of this contraint in rule T-Assign is to guarantee that, when a variable v can reach another variable v', the lifetime of v' is equal or larger than the lifetime of v. Otherwise, the deallocation of v' (at the end of its lifetime) would leave a dangling reference reachable from v.

Consider for instance the following illegal program.

$$\{\mathtt{let\ mut}\ x = \mathtt{box}\ 0;\ \mathtt{let\ mut}\ y = \&\mathtt{mut}\ *x;\ *x = 1\}^l$$

Let us apply the typing rules above starting from $\tau_1 = \{\}$.

- (T-Const) $\tau_1, l \vdash 0 : \mathsf{int} \dashv \tau_1$.
- (T-Box) $\tau_1, l \vdash \mathtt{box}\ 0 : \Box\, \mathsf{int} \dashv \tau_1$.
- (T-Declare) As $x \notin \mathsf{dom}(\tau_1)$, for $\tau_2 = \tau_1[x \to (\Box\, \mathsf{int})^l] = \{x \to (\Box\, \mathsf{int})^l\}$ we have $\tau_1, l \vdash \mathtt{let\ mut}\ x = \mathtt{box}\ 0 \dashv \tau_2$.
- (T-MutBorrow) By Def. 10, we have $\mathsf{type}(x, \tau_2) = \tau_2(x) = (\Box\, \mathsf{int})^l$, hence $\mathsf{type}(*x, \tau_2) = \mathsf{int}^l$, so $|\mathsf{type}(*x, \tau_2)| = \mathsf{int}$. Therefore, $|\mathsf{type}(*x, \tau)|$ is full because dangling does not occur in it. Moreover, $\neg\mathsf{writeProhibited}(*x, \tau_2)$ holds because $\mathsf{root}(*x) = x$ does not occur in a borrow nor in a mutable borrow inside τ_2. Finally, $\mathsf{mutable}(*x, |\tau_2(\mathsf{root}(*x))|, \tau_2) = \mathsf{mutable}(*x, |\tau_2(x)|, \tau_2) = \mathsf{mutable}(*x, \Box\, \mathsf{int}, \tau_2) = \mathsf{mutable}(x, \mathsf{int}, \tau_2) = \mathit{true}$. Consequently, we have $\tau_2, l \vdash \&\mathtt{mut}\ *x : \&\mathtt{mut}\ *x \dashv \tau_2$.
- (T-Declare) As $y \notin \mathsf{dom}(\tau_2)$, for $\tau_3 = \tau_2[y \to (\&\mathtt{mut}\ *x)^l]$, we have $\tau_2, l \vdash \mathtt{let\ mut}\ y = \&\mathtt{mut}\ *x \dashv \tau_3$.
- (T-Const) $\tau_3, l \vdash 1 : \mathsf{int} \dashv \tau_3$.
- (T-Assign) We have $\mathsf{write}(\tau_3, *x, \mathsf{int}) = \mathsf{apply}(x, \mathsf{update}(\tau_3, 1, |\tau_3(x)|, \mathsf{int})) = \mathsf{apply}(x, \mathsf{update}(\tau_3, 1, \Box\, \mathsf{int}, \mathsf{int}))$. Moreover, we have $\mathsf{update}(\tau_3, 1, \Box\, \mathsf{int}, \mathsf{int}) =$

$\mathsf{expand}(\mathsf{update}(\tau_3, 0, \mathsf{int}, \mathsf{int})) = \mathsf{expand}(\langle \tau_3, \mathsf{int} \rangle) = \langle \tau_3, \square\, \mathsf{int} \rangle$. Consequently, $\mathsf{write}(\tau_3, *x, \mathsf{int}) = \mathsf{apply}(x, \langle \tau_3, \square\, \mathsf{int} \rangle) = \tau_3[x \to (\square\, \mathsf{int})^l] = \tau_3$. However, $\neg\mathsf{writeProhibited}(*x, \tau_3)$ does not hold because $\mathsf{root}(*x) = x$ occurs in the mutable borrow `&mut` $*x$ inside τ_3. Therefore, (T-Assign) cannot be applied.

5 Termination

This section provides a sufficient condition for the termination of the typing algorithm for leftvalues in Def. 10. It is based on the idea that the Rust type system forces programmers to build *linear* data structures. This translates into a notion of *linearization* for typings, meaning that they map variables in a way that does not allow cycles: each variable is mapped into a type that only contains variables of strictly lower ranks.

The same condition, with a similar proof, can be used to prove that the other recursive functions used in the typing rules in Sec. 4 terminate, namely, mutable and write. The proof is identical and we have chosen type as a representative.

Definition 11. *A typing τ over a context κ is* linearizable *if there exists an injective function $\phi : \kappa \to \mathbb{N}$ such that, for every $x \in \kappa$ and every v that occurs in $\tau(x)$, it is $\phi(x) > \phi(v)$. We say that $\phi(y)$ is the* rank *of y.*

As an example, suppose $\kappa = \{x, y\}$ where $\tau = \{x \to \&y^l, y \to int^l\}$, then $\phi = \{x \to 0, y \to 1\}$ is a suitable linearisation. A linearizable typing induces an ordering between leftvalues: either the number of dereferences decreases, or the rank of their roots decreases.

Definition 12. *Given a context κ and a linearizable typing τ over κ, the relation $>$ between leftvalues is the minimal relation such that*

1. $*\mathsf{w} > \mathsf{w}$ *for every* $\mathsf{w} \in \mathsf{LV}_\kappa$*, and*
2. $\mathsf{w}_1 > \mathsf{w}_2$ *if* $\phi(\mathsf{root}(\mathsf{w}_1)) > \phi(\mathsf{root}(\mathsf{w}_2))$*, for every* $\mathsf{w}_1, \mathsf{w}_2 \in \mathsf{LV}_\kappa$.

Proposition 1. *The relation $>$ from Def. 12 is well-founded.*

Proof. Assume by contradiction that $>$ is not well-founded. Then there is an infinite sequence of leftvalues $s = \mathsf{w}_0 > \mathsf{w}_1 > \cdots > \mathsf{w}_n > \cdots$. Since, in the first rule of Def. 12, it is $\mathsf{root}(*\mathsf{w}) = \mathsf{root}(\mathsf{w})$ and consequently $\phi(\mathsf{root}(*\mathsf{w})) = \phi(\mathsf{root}(\mathsf{w}))$, we conclude that the rank of the root of the leftvalues decreases at most $|\kappa|$ times in s or remains constant. Hence, there is a finite k such that $\phi(\mathsf{root}(\mathsf{w}_k)) = \phi(\mathsf{root}(\mathsf{w}_{k+i}))$ for all $i \geq 0$. This means that, from k onwards, only rule 1 of Def. 12 applies. But that rule strictly decreases the size of the leftvalues and consequently cannot be applied indefinitely. This is incompatible with the hypothesis that s is infinite. □

Since $>$ is well-founded, it can be used in proofs by induction, as below.

Proposition 2. *If a typing τ over κ is linearizable, then the algorithm for computing* type *in Def. 10 terminates.*

Proof. We actually prove a stronger statement, namely that, given $\mathsf{w} \in \mathsf{LV}_\kappa$:

1. $\mathsf{type}(\mathsf{w}, \tau)$ terminates;
2. every variable v that occurs in $\mathsf{type}(\mathsf{w}, \tau)$ is such that $\phi(\mathsf{root}(\mathsf{w})) > \phi(v)$.

We proceed by induction on w. The base case is when w is actually the variable x of lowest rank. By Def. 10, it is $\mathsf{type}(x, \tau) = \tau(x)$ hence it terminates and no variable occurs in it, since (Def. 11) the rank of those variables should be even lower, which is impossible. Assume now that both 1 and 2 hold for all leftvalues w'' such that $\mathsf{w} > \mathsf{w}''$. If w is a variable x, then $\mathsf{type}(x, \tau) = \tau(x)$ hence $\mathsf{type}(\mathsf{w}, \tau)$ terminates and every variable v that occurs in $\tau(x)$ is such that $\phi(x) > \phi(v)$ (Def. 11). Hence both 1 and 2 hold for w as well. If, instead, $\mathsf{w} = *\mathsf{w}''$ for a suitable w'', then $\mathsf{w} > \mathsf{w}''$ (Def. 12) and by inductive hypothesis we know that 1 and 2 hold for w''. The computation of $\mathsf{type}(*\mathsf{w}'', \tau)$ first recurs on $\mathsf{type}(\mathsf{w}'', \tau)$ (Def. 10). In the first, second and third case of Def. 10, also the computation of $\mathsf{type}(*\mathsf{w}'', \tau)$ terminates and property 2 is vacuously true. In the sixth case of Def. 10, the computation of $\mathsf{type}(*\mathsf{w}'', \tau)$ terminates and $|\mathsf{type}(\mathsf{w}'', \tau)| = \square\,|\mathsf{type}(*\mathsf{w}'', \tau)|$. Since w'' and $*\mathsf{w}''$ have the same root, condition 2 lifts from w'' to $*\mathsf{w}''$. In the fourth and fifth case of Def. 10, by inductive hypothesis we know that 2 holds for w'' and consequently the root of w' in Def. 10 has lower rank than the root of w''. That is, $\mathsf{w}'' > \mathsf{w}'$. By inductive hypothesis, both 1 and 2 hold for w'. Hence $\mathsf{type}(\mathsf{w}', \tau)$ terminates and $\mathsf{type}(*\mathsf{w}'', \tau)$ terminates and yields $\mathsf{type}(\mathsf{w}', \tau)$. Every variable that occurs in $\mathsf{type}(\mathsf{w}', \tau)$ has lower rank than $\mathsf{root}(\mathsf{w}'') = \mathsf{root}(\mathsf{w})$. Therefore, both 1 and 2 hold for w also in this case. □

6 Semantical Invariant

This section proves an invariant of the typing rules from Sec. 4. Namely, it proves that if those rules are applied from a linearizable typing τ, they can only lead to a linearizable typing τ'. By Prop. 2, this means that the recursion used for typing leftvalues in those rules is well-founded, hence a borrow checker that implements those typing rules terminates (assuming that it starts from the empty, linearizable typing). The proof proceeds by rule induction.

Some rules from Sec. 4 obviously satisfy the invariant, since they do not modify the typing (for them, $\tau = \tau'$). This is the case of rules T-Const, T-Copy, T-ImmBorrow and T-MutBorrow. Rule T-Box satisfies the invariant by a simple application of rule induction.

For rule T-Move, it is $\tau' = \mathsf{move}(\mathsf{w}, \tau)$. The intuition is that strike can only make the set of variables in the right-hand side of the typing smaller. Therefore, it can never make τ' non-linearizable. This is proved below.

Lemma 1. *If T-Move is applied from a linearizable typing τ and leads to a typing τ', then also τ' is linearizable.*

Proof. By definition of move, the only difference between τ and τ' is at $r = \mathsf{root}(\mathsf{w})$. The variables that occur in $\tau'(r)$ are included in those that occur in $\tau(r)$ (strike can only strike away part of the type $\tau(r)$). Hence the same function ϕ that exists for τ (Def. 11) shows that τ' is linearizable. □

Rule T-Block is used at the end of a block of code, where the set S of local variables declared in the block goes out of scope. It removes the type bindings for the variables in S from the initial typing τ, through function drop. The invariant for T-Block follows by rule induction and by the following result, whose intuition is that the removal of bindings from a typing can never make it non-linearizable.

Lemma 2. *If drop is applied from a linearizable typing τ and leads to a typing τ', then also τ' is linearizable.*

Proof. The difference between τ and τ' is that τ' is missing some bindings for some variables that have been projected away. Therefore, the same function ϕ that exists for τ (Def. 11) can be used to show that τ' is linearizable. □

Rule T-Declare models the declaration of a new variable x, bound to an expression e. The evaluation of e leads to a typing τ' that, by rule induction, satisfies the invariant. As a final step, this rule enlarges τ' with a binding for x. Since x is fresh ($x \notin \mathsf{dom}(\tau)$), variable x does not occur in the right-hand side of that binding. Namely, the rule leads to a new typing $\tau'' = \tau'[x \to T^l]$ where T^l is the type of e, such that x does not occur in T. The invariant for T-Declare follows by the next result.

Lemma 3. *Let τ be a linearizable typing for the context κ; let $x \notin \kappa$, $T \in \mathsf{T}_\kappa$ (hence x does not occur in T) and l be a lifetime. Then $\tau' = \tau[x \to T^l]$ is linearizable as well.*

Proof. Consider the function ϕ that shows that τ is linearizable (Def. 11). Let us extend ϕ into an injective function ϕ' that gives x the highest rank:

$$\phi' = \phi \left[x \to 1 + \max_{y \in \kappa} \phi(y) \right].$$

Given $y \in \kappa$, it is $\phi'(y) = \phi(y) > \phi(v)$ for every v that occurs in $\tau(y) = \tau'(y)$. Since x is fresh, such v are distinct from x and we conclude that $\phi'(y) > \phi'(v)$ for every v that occurs in $\tau'(y)$. Since x does not occur in T, it is $\phi'(x) = 1 + \max_{y \in \kappa} \phi(y) > \phi(v) = \phi'(v)$ for every v that occurs in $T^l = \tau'(x)$. □

Rule T-Assign computes the type T of the value of the assigned expression e, which leads to a typing τ'. By rule induction, τ' is linearizable. Then the rule writes that value into a leftvalue w. It performs this by computing $\tau'' = \mathsf{write}(\tau', \mathsf{w}, T)$. The following result shows that τ'' is linearizable as well.

Lemma 4. *Let τ be a linearizable typing for the context κ; let $\mathsf{w} \in \mathsf{LV}_\kappa$ and $T \in \mathsf{T}_\kappa$. Let $\tau' = \mathsf{write}(\tau, \mathsf{w}, T)$ be the application of function write in rule T-Assign, used there to assign the type T to w. Then τ' is linearizable as well.*

Proof. The function write modifies a set of variables $v_1, \ldots, v_n$ in τ to compute τ'. The type of the other variables remains unchanged from τ to τ'. Since the type system guarantees that borrowed variables are not modified [25], this means

that $v_1, \ldots, v_n$ do not occur in the borrows in τ. Moreover, the variables in the borrows in T do not contain $v_1, \ldots, v_n$, because such variables are either $x = \mathsf{root}(\mathsf{w})$, and the rule T-Assign forbids the presence of x in the borrows in T ($\neg$writeProhibited in rule T-Assign); or they are inside mutable borrows in τ (last case of update), in which case they would be mutably borrowed and the type system would have forbidden to read mutably borrowed variables in order to compute the type T (see rule T-MutBorrow). This means that such $v_1, \ldots, v_n$ only occur in the left-hand side of the bindings of τ'. Consider now the function ϕ that shows that τ is linearizable (Def. 11). Let us extend ϕ into an injective function ϕ' that gives $v_1, \ldots, v_n$ the highest ranks:

$$\phi' = \phi \left[v_i \to i + \max_{y \in \kappa \setminus \{v_1, \ldots, v_n\}} \phi(y) \middle| 1 \leq i \leq n \right].$$

For every $y \in \kappa \setminus \{v_1, \ldots, v_n\}$, it is $\phi'(y) = \phi(x) > \phi(v) = \phi'(v)$ for every v that occurs in $\tau(y) = \tau'(y)$. Moreover, by construction, $\phi'(v_i) > \phi(v) = \phi'(v)$ for every v that occurs in $\tau'(v_i)$. That is, ϕ' is linearizable as well. □

7 Related Work

Reed provided an early formalisation of Rust called "Patina" which shares some similarities with FR [26]. For example, it employs a flow-sensitive type system for characterising borrow checking which operates over a "shadow" heap. However, the scope was significantly larger and, as such, soundness was not established. Likewise, Wang *et al.* presented a formal, executable operational semantics for Rust called KRust [32]. This was defined in $\mathbb{K}$ — a rewrite-based executable semantic framework particularly suited at developing operational semantics [27]. A large subset of Rust was defined in this way and partially validated against the official Rust test suite. Another example is that of Weiss *et al.*, who presented an unpublished system called *Oxide* which bears striking similarity with FR [33]. Oxide was also inspired by Featherweight Java to produce a relatively lean formalisation of Rust. Again, it includes a far larger subset of Rust than FR (perhaps making it more *middleweight* than *featherweight*). There are also differences, as Oxide doesn't model boxes explicitly and has no clear means to model heap allocated memory. The comprehensive work of Jung *et al.* provides a machine-checked formalisation for a realistic subset of Rust [10]. This includes various notions of concurrency and extends to libraries using `unsafe` features by identifying *library-specific verification conditions* which must be satisfied to ensure overall safety. However, concessions were understandably necessary given the enormity of this formalisation task (which, in fact, amounts to roughly 17.5KLOC of Coq). For example, the system presented does not resemble the surface syntax of Rust but, rather, is more akin to the *Mid-level Intermediate Representation (MIR)* used within the Rust compiler. Separately, Jung *et al.* explored compiler optimisations in the context of unsafe code [11]. This is challenging because, within unsafe code, the usual guarantees provided by Rust may not hold (*eg.*, multiple mutable borrows of the same location can exist). The proposed system, *Stacked*

Borrows, provides an operational semantics for memory accesses in Rust. This introduces a strong notion of *undefined behaviour* such that a compiler is permitted to ignore the possibility of such programs when applying optimisations (roughly in line with how C compilers handle undefined behaviour [19])

The potential hazards of `unsafe` code have been a considerable focus of academic work and, indeed, numerous bugs and security advisories have already been uncovered in real-world programs [4,34]. Interest has been growing in using state-of-the-art verification tools here. For example, Rudra employs a straightforward static analysis to scan for bug patterns related to error handling [4]. Nevertheless, the tool identified 74 new CVE's (including two in the standard library). Another good example is SMACK [7,8] which translates LLVM IR to Boogie/Z3 and was recently extended to Rust [6]. CRUST [30] is similar, but uses CBMC [12] as the backend. CRUST specifically focuses on memory safety violations (such as multiple mutable references to the same data). An interesting feature is support for automatically deriving "proof drivers" using a technique reminiscent of that for test case generation [22]. KLEE employs symbolic execution and was also extended to support Rust [16,17]. Unlike CRUST this tool considers a larger number of errors, including arithmetic overflow and buffer overruns (*ie.*, not just those related to memory unsafety). Prusti exploits automated theorem proving as the core technique, building on Viper [3]. This makes Prusti more comparable with tools such as Dafny [14,13] and Whiley [24,31] which require additional programmer annotations to verify memory-safety properties (*eg.* adding specifications to clarify method side-effects, etc). However, Prusti exploits aliasing information inherent in Rust programs to avoid much of this. Instead, programmers can focus on specifying properties of interest, such as the absence of arithmetic overflow or buffer overruns. Unfortunately, Prusti does not consider `unsafe` code (though it presumably could be managed with further specification). Finally, other relevant tools include Miri [21,11] (a partially symbolic interpreter for MIR) and RustHorn [18] (a specialised verifier based on Constrained Horn Clauses).

8 Conclusion

This paper has provided a proof of termination for the borrow checker of Featherweight Rust. As a consequence, it supports the use of that framework for the specification and analysis of the behaviour of Rust programs. The proof is based on the particular property of Rust, that imposes a strict discipline to programmers, so that only linearizable data strcutures can be constructed at run time. In this sense, the proof sheds more light on the reason of such design choice of the language.

References

1. B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the servo web browser engine using Rust. In *Proc. ICSE*, pages 81–89, 2016.

2. Mohammadreza Ashouri. Etherolic: A practical security analyzer for smart contracts. In *Proc. SAC*, page 353–356. ACM Press, 2020.
3. V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Proc. OOPSLA*, page Article 147, 2019.
4. Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim. Rudra: Finding memory safety bugs in Rust at the ecosystem scale. page (to appear), 2021.
5. A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, and L. Ryzhyk. System programming in Rust: Beyond safety. *OS Review*, 51(1):94–99, 2017.
6. M. Baranowski, S. He, and Z. Rakamarić. Verifying Rust programs with SMACK. In *Proc. ATVA*, pages 528–535, 2018.
7. M. Barnett, B. Evan Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. FMCO*, pages 364–387, 2006.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.
9. F. Þ. Hjálmarsson, G. K. Hreiðarsson, M. Hamdaqa, and G. Hjálmtýsson. Blockchain-based e-voting system. In *Proc. CLOUD*, pages 983–986, 2018.
10. R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proc. POPL*, pages 1–34, 2018.
11. Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. In *Proc. POPL*, page Article 41, 2020.
12. Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *Proc. TACAS*, pages 389–391. Springer-Verlag, 2014.
13. K. R. M. Leino. Developing verified programs with Dafny. In *Proc. VSTTE*, pages 82–82, 2012.
14. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, pages 348–370, 2010.
15. A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis. The case for writing a kernel in Rust. In *Proc. APSYS*, pages 1:1–1:7, 2017.
16. M. Lindner, J. Aparicius, and P. Lindgren. No panic! verification of Rust programs by symbolic execution. In *Proc. INDIN*, pages 108–114, 2018.
17. M. Lindner, N. Fitinghoff, J. Eriksson, and P. Lindgren. Verification of safety functions implemented in Rust - a symbolic execution based approach. In *Proc. INDIN*, pages 432–439, 2019.
18. Y. Matsushita, T. Tsukada, and N. Kobayashi. RustHorn: CHC-based verification for Rust programs. In *Proc. ESOP*, pages 484–514, 2020.
19. K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. In *Proc. POPL*, pages 67:1–67:32, 2019.
20. Pengxiang Ning and Boqin Qin. Stuck-me-not: A deadlock detector on blockchain software in Rust. *Procedia Computer Science*, 177:599–604, 2020.
21. Scott Olson. Miri: an interpreter for Rust's mid-level intermediate representation. Technical report, 2016.
22. Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *Proc. OOPSLA (Companion)*, pages 815–816, 2007.
23. D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.
24. D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing Whiley. *SCP*, pages 191–220, 2015.

25. David J. Pearce. A lightweight formalism for reference lifetimes and borrowing in Rust. *ACM TOPLAS*, 43(1):Article 3, 2021.
26. Erik Reed. Patina: A formalization of the Rust programming language. Technical report, 2015.
27. G. Rosu and T. Serbanuta. An overview of the K semantic framework. *JLAP*, 79(6):397–434, 2010.
28. Rust Team. The Rust programming language. `doc.rust-lang.org/book/`. Retrieved 2016-05-01.
29. Rust Team. The rustonomicon - the dark arts of advanced and unsafe Rust programming. `doc.rust-lang.org/nomicon/`. Retrieved 2020-31-03.
30. J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for Rust. In *Proc. ASE*, pages 75–80, 2015.
31. M. Utting, D. J. Pearce, and L. Groves. Making Whiley Boogie! In *Proc. IFM*, pages 69–84, 2017.
32. F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. Krust: A formal executable semantics of Rust. In *Proc. TASE*, pages 44–51, 2018.
33. A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed. Oxide: The essence of rust, 2019.
34. H. Xu, Z. Chen, M. Sun, and Y. Zhou. Memory-safety challenge considered solved? an empirical study with all Rust cves. *CoRR*, abs/2003.03296, 2020.
35. F. Zhang, W. He, R. Cheng, J. Kos, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. The ekiden platform for confidentiality-preserving, trustworthy, and performant smart contracts. *IEEE S&P*, 18(3):17–27, 2020.