Stacked Borrows: An Aliasing Model for Rust

RALF JUNG, Mozilla, USA and MPI-SWS, Germany
HOANG-HAI DANG, MPI-SWS, Germany
JEEHOON KANG, KAIST, Korea
DEREK DREYER, MPI-SWS, Germany

Type systems are useful not just for the safety guarantees they provide, but also for helping compilers generate more efficient code by simplifying important program analyses. In Rust, the type system imposes a strict discipline on pointer aliasing, and it is an express goal of the Rust compiler developers to make use of that alias information for the purpose of program optimizations that reorder memory accesses. The problem is that Rust also supports unsafe code, and programmers can write unsafe code that bypasses the usual compiler checks to violate the aliasing discipline. To strike a balance between optimizations and unsafe code, the language needs to provide a set of rules such that unsafe code authors can be sure, if they are following these rules, that the compiler will preserve the semantics of their code despite all the optimizations it is doing.

In this work, we propose *Stacked Borrows*, an operational semantics for memory accesses in Rust. Stacked Borrows defines an aliasing discipline and declares programs violating it to have *undefined behavior*, meaning the compiler does not have to consider such programs when performing optimizations. We give formal proofs (mechanized in Coq) showing that this rules out enough programs to enable optimizations that reorder memory accesses around unknown code and function calls, based solely on intraprocedural reasoning. We also implemented this operational model in an interpreter for Rust and ran large parts of the Rust standard library test suite in the interpreter to validate that the model permits enough real-world unsafe Rust code.

CCS Concepts: • **Theory of computation → Operational semantics**.

Additional Key Words and Phrases: Rust, operational semantics, alias analysis, program transformation

ACM Reference Format:
Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (January 2020), 32 pages. https://doi.org/10.1145/3371109

1 INTRODUCTION

Type systems are useful not only for making our programs more secure and reliable, but also for helping compilers generate more efficient code. For example, a language with a strong type system does not have to waste time and space maintaining dynamic type information on all run-time data. In a language like Rust [Klabnik and Nichols 2018], where the type system imposes a strict discipline on pointer aliasing, it is thus a natural question to ask whether that static discipline can be exploited by the compiler.

In particular, *mutable references* `&mut T` in Rust are unique pointers that cannot alias with anything else in scope. This knowledge should enable us to optimize the following function:

Authors' addresses: Ralf Jung, Mozilla, USA and MPI-SWS, Saarland Informatics Campus, Germany, jung@mpi-sws.org; Hoang-Hai Dang, MPI-SWS, Saarland Informatics Campus, Germany, haidang@mpi-sws.org; Jeehoon Kang, KAIST, Korea, jeehoon.kang@kaist.ac.kr; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

2475-1421/2020/1-ART41
https://doi.org/10.1145/3371109

```
1  fn example1(x: &mut i32, y: &mut i32) -> i32 {
2    *x = 42;
3    *y = 13;
4    return *x; // Has to read 42, because x and y cannot alias!
5  }
```

Since mutable references are unique, `x` and `y` cannot alias. As a consequence, the compiler should be allowed to assume that the read in line 4 will yield 42, so it can remove the memory access and replace it by a constant. Typically, the way such an optimization would be justified is through *reordering* of instructions. If we know `x` and `y` do not alias, then we can reorder lines 2 and 3, and it becomes clear that the value read from `x` in line 4 must be the value just written to it, namely 42.

Having access to this kind of alias information is a compiler writer's dream, in part because it is essential in justifying reorderings and other program transformations which are key to improving code generation [Ghiya et al. 2001; Wilson and Lam 1995], and in part because such alias information is typically hard to come by [Horwitz 1997]. In particular, the alias analysis that is possible in most programming languages is fundamentally limited by the weakness of conventional type systems. For example, in cases like the one above, where `x` and `y` are passed in to a function from its environment, a C/C++ compiler can make no *local* assumption about whether these pointers alias with each other or with any other pointer the program might be using—it would need to perform a more global, interprocedural analysis, which may be prohibitively expensive (or impossible if the whole program is not available). In contrast, because it enforces a strict discipline of pointer usage, the Rust type system provides a rich source of alias information that can be used to justify transformations like the one above *intraprocedurally*.

Unfortunately, there's a catch: Rust supports **`unsafe`** code. It is easy to write **`unsafe`** code that, when compiled with the current compiler[1], makes the above function return 13:

```
1  fn main() {
2    let mut local = 5;
3    let raw_pointer = &mut local as *mut i32;
4    let result = unsafe { example1(&mut *raw_pointer, &mut *raw_pointer) };
5    println!("{}", result); // Prints "13".
6  }
```

In line 3, this code uses the **`as`** operator to cast a *reference* to `local` (that would have type `&mut i32`) into a *raw pointer* with type `*mut i32`. Raw pointers are not tracked by the type system: they can be freely interconverted with integers in safe code, and arbitrary address arithmetic is possible. However, to maintain soundness of the type system, *dereferencing* a raw pointer is only permitted inside **`unsafe`** blocks, which serve as syntactic markers that the programmer opted-in to potentially unsafe behavior. Raw pointers are necessary when interfacing with C code through FFI, and also when implementing low-level pointer-heavy data structures in Rust that internally do not follow Rust's aliasing discipline.

In the example above, though, we are using raw pointers for a more sinister purpose: in line 4, we convert the raw pointer back to a reference. `&mut *raw_pointer` dereferences the raw pointer and immediately takes the address again, so this is effectively a cast from `*mut i32` back to `&mut i32`. The sinister part about this cast is that we do it twice! The type system does not stop us, as it does not even attempt to track what happens with raw pointers. As a result of all of this, we call `example1` with two aliasing references that both point to `local`, and the function returns 13.

It is tempting to ignore this problem because it "just" concerns **`unsafe`** code. However, that would be neglecting the important role **`unsafe`** code plays in the Rust ecosystem. Not all Rust code

[1] All tests were done with the Rust stable release 1.35.0 in release mode.

is safe—in fact pretty much all programs depend on *some* `unsafe` code—but that does not mean that Rust's safety guarantees are useless. What is important is that `unsafe` code is *encapsulated within a safe abstraction*. This approach lets Rust programmers use data structures such as `Vec` and `HashMap` (which are implemented using `unsafe` code) without worrying about memory safety issues, while at the same time not compromising efficiency when compared with unsafe languages such as C or C++. The Rust ecosystem rests on the interplay of safe code where possible and `unsafe` code where needed, and hence any realistic consideration of compilation for Rust programs requires proper treatment of `unsafe` code.

So, let us look at our example program again. (Together, `main` and `example1` form a closed program.) We want to argue that it is correct to compile this program in a way that prints `42`, but the only way we can make such an argument is by tweaking the operational semantics! The fact that the program "circumvents" the type system is irrelevant, because the type system is not involved in the definition of what it means for a Rust compiler to be correct. A compiler correctness statement that only applies to well-typed programs (in the safe fragment of Rust) would be useless for all practical purposes, because—as argued above—most programs being compiled contain `unsafe` code. So, with the type system being ruled out, the only knob we have left is the *dynamic semantics* of the source language: we have to define the behavior of our source program in a way that actually, it *is* allowed for the program to output `42`. More precisely, we will define the operational semantics in such a way that our example program has *undefined behavior*, which means the compiler is allowed to compile it in any way it chooses.

This is not a simple task. A naive semantics, such as the one used in RustBelt [Jung et al. 2018], will give the example program a defined meaning and thus force the compiler to print `13`. Compared to RustBelt's semantics, we have to "add" some undefined behavior to obtain the desired optimizations. But of course we should not add "too much" undefined behavior! We have to be careful that "desired" programs are still well-defined. This includes all safe programs, but should also include enough `unsafe` programs to still make `unsafe` Rust a useful language for implementing data structures, and to minimize the chances of programmers accidentally running into undefined behavior.

In this paper, we propose *Stacked Borrows*: an operational semantics for Rust that enforces Rust's aliasing discipline. It does so by introducing clearly-defined conditions under which a Rust program exhibits undefined behavior due to an aliasing violation. The key idea is to define a dynamic version of the static analysis—called the *borrow checker*—which Rust already uses to check that references are accessed in accordance with the aliasing discipline. The borrow checker enforces in particular that references that might be overlapping are used in a well-nested manner. We model this discipline in our dynamic analysis with the use of a per-location *stack*: our dynamic analysis detects when references get used in a non-stack-like way, and flags such programs as having undefined behavior. Based on that core structure, we then extend Stacked Borrows with rules for raw pointers (which the borrow checker ignores) with the goal of being maximally liberal while not interfering with the key properties of the "safe fragment" of our dynamic analysis.

We have validated Stacked Borrows in two ways:

- To ensure that Stacked Borrows does not introduce "too much" undefined behavior, we have equipped Miri, an existing interpreter for Rust programs, with an implementation of Stacked Borrows. We have run the OS-independent part of the Rust standard library test suite[2] in this interpreter. In so doing, we uncovered a few violations of the model, almost all of which have been accepted by the Rust developers as bugs and have since been fixed in the standard library. The majority of tests passed without any adjustments.

[2]Concretely, the tests for `libcore`, `liballoc`, and the `HashMap`.

- To ensure that Stacked Borrows has "enough" undefined behavior, we give proof sketches demonstrating that a few representative compiler transformations, such as the one in `example1`, are legal under this semantics. These proof sketches are all backed by fully mechanized proofs in Coq, based on a formalization of Stacked Borrows and a framework for open simulations in the style of Hur et al. [2012].

In future work, we would like to connect this work with RustBelt, by showing that the RustBelt type safety and library correctness proofs still hold under this stricter operational semantics (stricter in that it imposes more restrictions on what constitutes well-defined behavior). We also would like to expand our formal verification to consider concurrency; we expect the approach to scale to concurrency, but for now we are just considering the sequential case.

Overview. In §2, we begin with an introduction to Rust, explaining what is relevant concerning the type system and its handling of references. We go on in §3 to introduce a basic version of Stacked Borrows and give a proof sketch for (a generalized version of) the desired optimization for `example1` above. We then extend this basic model in §4 to support more optimizations and in §5 to support more Rust features. Section §6 consolidates the prior discussion with a formal definition of Stacked Borrows. Finally, in §7 we describe our evaluation of Stacked Borrows with Miri, before discussing related work in §8 and concluding in §9.

2 AN INTRODUCTION TO RUST

To understand the remainder of this paper, some basic knowledge of the Rust type system is required, in particular its rules for references. This section serves to lay the necessary foundations.

The key principle of the Rust type system that we will focus on here is the following *exclusion principle*: data can *either* be mutated through *one* reference, *or* it can be *immutably* shared amongst many parties—but not both at the same time. (There is an exception to this that we will come back to later.) The motivation for this rule is to ensure memory safety. Consider the following example:

```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // Points *into* v.
3  v.push(12); // May reallocate the backing store of v.
4  println!("v[1] = {}", *vptr); // Compiler error!
```

Here, `v` is a heap-allocated array of integers (a `Vec<i32>`, corresponding to `std::vector<int>` in C++). `vptr` is a reference of type `&mut i32` pointing *into* the allocation where `v` stores its data. Such *interior pointers* are generally not supported in garbage-collected languages, but they are widely used in languages such as C, C++, and Rust to achieve a more compact and cache-friendly layout of data structures.

The problem with interior pointers becomes apparent in line 3 of the example: `v.push` might have to allocate new space elsewhere on the heap if there is not enough space for a third element in the original location. By mutating `v`, we inadvertently deallocated the memory that `vptr` points to. This is why, in line 4, the Rust compiler refuses to compile this program, complaining that we "cannot borrow `v` as mutable more than once at a time".

The error talks about "borrowing", but before we can explain that, we have to talk about *ownership*. In Rust, data always has a *unique owner*. For example, when `v` gets created in line 1, the `Vec` is fully owned by our example function. The Rust type system tracks ownership and ensures that when a variable is passed to a function, ownership is passed along, so that the caller may no longer access that data. Because ownership is unique, owned data can be arbitrarily mutated without violating the exclusion principle.

But a function like `push` is supposed to work on `v` without consuming the vector. To implement this, Rust introduces the idea of *borrowing* data, which is triggered by creating a reference (using the

&mut or & operator). We cannot see a reference being created in line 3, but that is just because the compiler applies some type-based syntactic sugar: line 3 desugars to `Vec::push(&mut v, 12)`. So the first argument does not get passed by value (which would be interpreted as passing ownership). Rather, it gets passed by reference, which is interpreted as *temporarily lending* `v` to `push`—or, in other words, `push` *borrows* `v` from its caller.

However, the reference in line 3 is not the only one being created in this short program. In line 2, we also create a reference: `vptr` borrows from `v` as well. To ensure the exclusion principle, the compiler has to make sure that the same data is not borrowed twice at the same time. This analysis rests on the concept of a *lifetime*: as in real life, when lending something, frustration and misunderstanding can be prevented by agreeing up front on how long something may be borrowed. To this end, when a reference gets created, it gets assigned a lifetime. Moreover, the compiler notes that there exists a *loan* of the borrowed-from data (`v[1]` in our case) that carries the same lifetime. (This lifetime is almost always inferred, but then both the reference and the loan use the same inference variable.) This is enough information to ensure

(1) that the reference can only be used while its lifetime is ongoing, and
(2) that the original referent does not get used until the lifetime of the loan has expired.

The analysis checking these properties is called *borrow checking*.

In our example program, the compiler infers the following lifetimes (`&'a mut v` is not actual Rust syntax, we just use it for illustration purposes):

```
1  let mut v = vec![10, 11];
2  let vptr = &'a mut v[1];
3  Vec::push(&'b mut v, 12);          Lifetime 'b
4  println!("v[1] = {}", *vptr);              Lifetime 'a
```

The lifetime `'b` for the second borrow just spans the call of `push`; there are no further restrictions, so the compiler makes `'b` as short as possible. However, in line 4 we use the reference `vptr` with lifetime `'a`, so to satisfy rule (1) we are forced to extend `'a` until line 4. But this means that when a reference to `v` gets created in line 3 (which counts as a "use" of `v`), we still have an outstanding loan of `v[1]` for lifetime `'a`! This violates condition (2) of that loan, so the program gets rejected.

This also explains why reordering lines 2 and 3 of the original example fixes the error:

```
1  let mut v = vec![10, 11];
2  Vec::push(&'b mut v, 12);          Lifetime 'b
3  let vptr = &'a mut v[1];
4  println!("v[1] = {}", *vptr);     Lifetime 'a
```

Now each time a new reference gets created, there is no outstanding loan: the loan of `v` for lifetime `'b` ends before `vptr` gets created. Hence the borrow checker accepts this program.

Reborrowing. From the above example, it may seem like lifetimes of mutable references can never overlap, but that is not actually the case: when we create a reference from an existing reference, their lifetimes will actually be *nested*. For example:

```
1  let mut v = vec![10, 11];
2  let v2 = &'a mut v;
3  let vptr = &'b mut (*v2)[1];
4  println!("v[1] = {}", *vptr);       Lifetime 'b
5  Vec::push(v2, 12);                          Lifetime 'a
```

Here, `vptr` gets *derived from* `v2`. This operation is also called a *reborrow*. To handle reborrowing, condition (1) from above gets extended to say that both the reference with lifetime `'a` *and all*

references derived from it may only be used while `'a` is ongoing. Practically speaking, this means that the lifetime `'b` of the "inner" reference `vptr` must be included in the lifetime `'a` of the "outer" reference `v2`.

In this example, adding another use of `vptr` at the end would introduce a bug: pushing to `v2` in line 5 might invalidate `vptr`, just like in our very first example! Rust statically detects such a bug just as before: when creating `vptr`, a loan gets attached to `v2` remembering that this reference has been "given away" and may not be used until `'b` is over. So if `'b` were to extend past line 5, then in line 5 the borrow checker would detect that `v2` is being used while there exists an outstanding loan of it, and that would be flagged as an error for violating condition (2).

Shared references. So far, we have only considered *mutable* references, written `&mut T`. To also cover the other disjunct of the exclusion principle, Rust features *shared references*, written `&T`. Shared references permit aliasing, but no mutation.

Many shared references may co-exist at the same time, with overlapping (non-nested) lifetimes:

```
1  let mut v = vec![10, 11];
2  let vptr = &'a v[1];
3  let vptr2 = &'b v[1];
4  println!("v[1] = {}", *vptr);          Lifetime 'a
5  println!("v[1] = {}", *vptr2); Lifetime 'b
```

Here, we create two shared references to the same element of `v`, and the calls to `println` demonstrate that they can both be used in an interleaved fashion.

The borrow checker performs almost the same checks as for mutable references, except that condition (2) gets weakened to ensuring that the referent does not get *mutated* until the lifetime of the loan has expired. So in line 2, a *shared* loan with lifetime `'a` gets attached to `v[1]`, which is still outstanding when `v[1]` gets used again in line 3. However, creating another shared reference to `v[1]` counts as a non-mutating use of it, and thus there is no conflict between the loan and the operation. Thus we just add a second loan to `v[1]`, this time for lifetime `'b`. Only once *both* loans have expired may `v[1]` be mutated again.

If we added a mutation of `v` (say, `v.push(12)`) between lines 3 and 4, the program would get rejected: such a mutation would be in conflict with the outstanding shared loans with lifetimes `'a` and `'b`. The compiler will also reject any attempt to write through a shared reference.

Raw pointers. Sometimes, the aliasing discipline for references that the borrow checker enforces is too strict. One typical example of this is a data structure where the pointers form a cycle: this leads to there being many paths to the same data (*i.e.*, there is aliasing), and hence the borrow checker will rule out all mutation. In this case, one option is to not use references but instead employ *raw pointers*. Raw pointers in Rust basically act like pointers in C: they are not checked by the compiler, and there are no aliasing restrictions. For example, the following code demonstrates a legal way to cause aliasing with raw pointers:

```
1  let mut x = 42;
2  let ptr1 = &mut x as *mut i32; // create first raw pointer
3  let ptr2 = ptr1; // create aliasing raw pointer
4  // Write through one pointer, observe with the other pointer.
5  unsafe { *ptr1 = 13; }
6  unsafe { println!("{}", *ptr2); } // Prints "13".
```

In line 2, we cast a reference to raw pointer type. Raw pointers can be duplicated (there is no ownership attached to them), so we can just copy that pointer in line 3. Then we can use both pointers despite their aliasing, just as we would in C. However, we need to put these uses of

raw pointers inside an **unsafe** block to acknowledge that we did our due diligence and manually ensured safety of all involved operations: since the borrow checker does not do any tracking for raw pointers, the compiler cannot itself guarantee that dereferencing them will not cause problems.

3 STACKED BORROWS, PART I

The key idea of Stacked Borrows is to take the static analysis that the borrow checker performs, which uses lifetimes, and develop from it a *dynamic analysis* that does *not use lifetimes*. Then we can require that even programs using **unsafe** code must satisfy this dynamic analysis, which gives the compiler license to assume that as a fact during optimization. Safe programs should trivially satisfy the dynamic analysis, because it is meant to be strictly more liberal than the borrow checker.

Avoiding the use of lifetimes in Stacked Borrows was a deliberate choice. Lifetimes are in many cases *inferred* by the compiler, even in **unsafe** code, and whether that code is considered well-behaved should not depend on the mercurial details of the compiler's lifetime inference. In particular, this inference has recently undergone significant change—with the switch from the old AST-based borrow checker to non-lexical lifetimes ("NLL") [Klock 2019]—and it is planned to change again as part of an ongoing project called "Polonius" [Matsakis 2018]. Such changes should not affect the way people have to think about their unsafe code. Moreover, lifetimes are erased from the program during the Rust compilation process immediately after borrow checking is done, so the optimizer does not even have access to that information. Thus practically speaking, making Stacked Borrows depend on lifetimes would not be useful for analyses performed by the compiler.

We will build up the Stacked Borrows semantics incrementally, beginning in the simplest possible context: suppose Rust had neither shared references nor raw pointers, only mutable references.

3.1 Mutable References in a Stack

In this simplified language, what does a dynamic version of the borrow checker look like? To reiterate, the borrow checker ensures

(1) that a reference and all references derived from it can only be used during its lifetime, and
(2) that the referent does not get used until the lifetime of the loan has expired.

We can re-phrase this property without referring to lifetimes, by saying that *every use of the reference (and everything derived from it) must occur before the next use of the referent (after the reference got created)*. For reasons that will become clear shortly, we call this the "stack principle". This is equivalent to what the borrow checker does, if we think of the lifetime as ending some time between the last use of the reference and the next use of the referent.

So, let us consider the following example:

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

This program violates the stack principle because a use of the reference `y` occurs in line 5 *after* the next use of referent `x` in line 4 ("next" after `y` has been created). It hence gets rejected by the borrow checker.[3]

If we look at the usage patterns of variables here, we can see that we start by creating `x`, then we create `y`, then we use `x` again—and then, and that is the point where the code violates the stack

[3]For simple integer types rejecting this program might seem silly, but the analysis is supposed to work for all types, and in the previous section we have seen how programs can go wrong if we use more complex types such as `Vec<i32>` where a mutation can invalidate existing pointers.

$$t \in \mathit{Tag} \triangleq \mathbb{N} \qquad \mathit{Scalar} \triangleq \mathsf{Pointer}(\ell, t) \mid z \quad \text{where } z \in \mathbb{Z}$$
$$\mathit{Item} \triangleq \mathsf{Unique}(t) \mid \ldots \qquad \mathit{Mem} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Scalar} \times \mathit{Stack}$$
$$\mathit{Stack} \triangleq \mathit{List}(\mathit{Item})$$

Fig. 1. Stacked Borrows domains (preliminary)

principle, we use y again. This demonstrates that the stack principle enforces a *well-nested* usage of references: the use of the derived reference y must be "nested between" other uses of x and cannot be arbitrarily interleaved. The "XYXY" sequence of usages violates this idea of nesting. As always when things are well-nested, that indicates a *stack discipline*, and indeed we will think of these aliasing references as being organized on a stack.

At a high level, the way Stacked Borrows rejects this program is by tracking a stack of references that are allowed to access `local`. The stack tracks which borrows of `local` may still be used, so we also call it the *borrow stack* to disambiguate it from, *e.g.*, the call stack. Newly created references get pushed on the top of the borrow stack, and we enforce that *after* a reference gets used, it is again at the top of the stack. To this end, all references above it get popped, which permanently invalidates them—references not in the stack may not be used at all. So, line 3 pushes y on top of x in that stack, but line 4 removes y again because we used x. Thus, in line 5, y is no longer in the stack, and hence its use is found to be a violation of the stack principle, and therefore undefined behavior.

3.2 An Operational Model of the Borrow Checker

To make this idea more precise, we first have to be able to distinguish different references that point to the same memory. x and y in the previous example program point to the same location, but to explain what is going on we have to distinguish them. So we assume that every reference gets *tagged* by some unique "pointer ID" t that gets picked when a reference gets created, and is preserved as the reference gets copied around. Formally (see Figure 1), we say that a *pointer value* $\mathsf{Pointer}(\ell, t)$ consists of a location ℓ in memory that the pointer points to, and a tag t identifying the pointer/reference. Both references and raw pointers are represented as pointer values at run-time.

In memory, we then store for every location ℓ a *borrow stack* of tags identifying the references that are allowed to access this location. To prepare for future extensions of the model in the following sections, we call the elements of the stack *items*; so far, $\mathsf{Unique}(t)$ holding a tag t is the only kind of item. Of course we also store the value that the memory holds at this location; in our case, such primitive values (integers and pointers) are called *scalars*.

The operational semantics acts on the tags and borrow stacks as follows:

Rule (NEW-MUTABLE-REF)**.** Any time a new mutable reference gets created (via `&mut expr`) from some existing pointer value $\mathsf{Pointer}(\ell, t)$ (the referent), first of all this is considered a *use* of that pointer value (so we follow USE-1 below). Then we pick some fresh tag t'. The new reference gets value $\mathsf{Pointer}(\ell, t')$, and we push $\mathsf{Unique}(t')$ on top of the stack for ℓ.

Rule (USE-1)**.** Any time a pointer value $\mathsf{Pointer}(\ell, t)$ gets *used*, an item with tag t must be in the stack for ℓ. If there are other tags above it, pop them, so that the item with tag t is on top of the stack afterwards. If $\mathsf{Unique}(t)$ is not in the stack at all, this program has undefined behavior.

These rules reflect the stack principle: a newly created reference may only be used as long as its item is in the stack, which is until the next time the referent it got *created from* gets used.

We can see these rules in action in the following annotated version of the previous example, where we spell out which reference gets which pointer value at run-time, and how the borrow stack of the memory storing `local` evolves over time. We assume that `local` is allocated at address ℓ, and $h \in \mathit{Mem}$ refers to the current memory at the given program point. Direct accesses to a local variable also use a tag, we assume that tag is 0 here.

```
1  let mut local = 42; // Stored at location ℓ, and with tag 0.
2  // The initial stack: h(ℓ) = (42, [Unique(0)])
3  let x = &mut local; // = Pointer(ℓ, 1)
4  // Use local, push tag of x (1) on the stack: h(ℓ) = (42, [Unique(0), Unique(1)]). (NEW-MUTABLE-REF)
5  let y = &mut *x; // = Pointer(ℓ, 2)
6  // Use x, push tag of y (2): h(ℓ) = (42, [Unique(0), Unique(1), Unique(2)]). (NEW-MUTABLE-REF)
7  *x += 1;
8  // Remove tag of y (2) to bring tag of x (1) to the top: h(ℓ) = (43, [Unique(0), Unique(1)]). (USE-1)
9  *y = 2;
10 // Undefined behavior! Stack principle violated: tag of y (2) is not in the stack. (USE-1)
```

3.3 Accounting for Raw Pointers

We are almost ready to come back to the example from the introduction. Remember, our goal is to explain how and where that program violates the "dynamic borrow checker" that is Stacked Borrows. We need this program to be a violation of Stacked Borrows, because it would otherwise constitute a counterexample to the desired optimization of `example1`.

So far, however, our model just deals with mutable references. We have to explain what happens when raw pointers get created (via `expr as *mut T`). Just like the borrow checker does not do any tracking for raw pointers, Stacked Borrows also makes no attempt to distinguish different raw pointers pointing to the same thing: raw pointers are *untagged*. So we extend our set *Tag* of tags with a $\bot$ value to represent untagged pointers.

Just like mutable references, raw pointers get added to the borrow stack. The idea is that an "XYXY" is allowed when *both* "X" and "Y" are raw pointers, but if either one of them is a mutable reference, we want that to still be a violation of the stack principle. This way, when performing analyses for compiler optimizations, we can be sure that mutable references are never part of an "XYXY" pattern of memory accesses.

To track raw pointers in the borrow stack, we add a second kind of items that can live in the stack: SharedRW (short for "shared read-write") items indicate that the location has been "shared" and is accessible to all raw (untagged) pointers for reading and writing. Overall, we now have:

$$t \in \mathit{Tag} \triangleq \mathbb{N} \cup \{\bot\} \qquad \mathit{Item} \triangleq \mathsf{Unique}(t) \mid \mathsf{SharedRW} \mid \ldots$$

We amend the operational semantics as follows, where USE-2 replaces USE-1:

Rule (NEW-MUTABLE-RAW-1)**.** Any time a mutable raw pointer gets created via a cast (`expr as *mut T`) from some mutable reference (`&mut T`) with value $\mathsf{Pointer}(\ell, t)$, first of all this is considered a *use* of that mutable reference (see USE-2). Then the new raw pointer gets value $\mathsf{Pointer}(\ell, \bot)$, and we push SharedRW on top of the borrow stack for ℓ.

Rule (USE-2)**.** Any time a pointer value $\mathsf{Pointer}(\ell, t)$ gets used, if t is $\bot$ then SharedRW must be in the stack for ℓ; otherwise $\mathsf{Unique}(t)$ must be in the stack. If there are other tags above that item, pop them. If the desired item is not in the stack at all, we found a violation of the stack principle.

Note how for tagged pointer values (*i.e.*, mutable references), USE-2 is the same as USE-1.

With this, the example program from the introduction executes as follows (we reordered the functions so that this can be read top-to-bottom):

```
1  fn main() {
2    let mut local = 5; // Stored at location ℓ, and with tag 0, h(ℓ) = (5, [Unique(0)]).
3    let raw_pointer = &mut local as *mut i32; // = Pointer(ℓ, ⊥)
4    // The temporary reference gets tag 1, and gets pushed: h(ℓ) = (5, [Unique(0), Unique(1)]).
5    // Then the raw pointer gets pushed: h(ℓ) = (5, [Unique(0), Unique(1), SharedRW]).
6    // (NEW-MUTABLE-REF, NEW-MUTABLE-RAW-1)
7    let result = unsafe { example1(
8      &mut *raw_pointer, // = Pointer(ℓ, 2)
9      // First reference gets added on top of the raw pointer: h(ℓ) = (5, [..., SharedRW, Unique(2)]).
10     // This uses the raw pointer! (NEW-MUTABLE-REF)
11     &mut *raw_pointer // = Pointer(ℓ, 3)
12     // Using raw_pointer here pops the first reference off the stack: h(ℓ) = (5, [..., SharedRW, Unique(3)]).
13     // This uses the raw pointer! (NEW-MUTABLE-REF)
14   ) }; // Next: jump to example1 (line 17).
15   println!("{}", result); // Prints "13".
16 }
17 fn example1(x: &mut i32, y: &mut i32) -> i32 {
18   // x = Pointer(ℓ, 2), y = Pointer(ℓ, 3), h(ℓ) = (5, [Unique(0), Unique(1), SharedRW, Unique(3)])
19   *x = 42;
20   // Analysis error! Tag of x (which is 2) is not in the stack. Program has undefined behavior.
21   *y = 13;
22   return *x; // We want to optimize this to return the constant 42.
23 }
```

The key point in this execution is when the two references that get passed to `example1` are created: each time we execute `&mut *raw_pointer`, NEW-MUTABLE-REF says this is *using* `raw_pointer` and as such we make sure that SharedRW is at the top of the borrow stack (USE-2). When creating the second reference (later called `y`), we have to pop off Unique(2), rendering the first reference (`x`) unusable! When `x` *does* get used in line 19, this is detected as a violation of the stack principle.

It may seem strange that a reborrow of a reference (as opposed to actually accessing memory) counts as a "use" of the old reference (the operand of the reborrow), but that is important: if we did not do so, after the second `&mut *raw_pointer`, we would end up with the following stack for ℓ:

$$[\text{Unique}(0), \text{Unique}(1), \text{SharedRW}, \text{Unique}(2), \text{Unique}(3)]$$

This borrow stack encodes that the reference tagged 3 is somehow "nested within" the reference tagged 2, and 3 may only be used until 2 is used the next time—which are the rules we want when 3 is reborrowed from 2, but that was not the case! To make sure that the stack adequately reflects which pointer values were created from which other pointer values, we make merely creating a reference "count" as a write access of the old reference (the operand). Thus, the stack will instead be $[\ldots, \text{SharedRW}, \text{Unique}(3)]$, which encodes the fact that 3 was created from a raw pointer and not from 2. In future work, we would like to explore other models that track the precise pointer inheritance information in a *tree*, instead of a stack.

3.4 Retagging, and a Proof Sketch for the Optimization on Mutable References

The next step in developing Stacked Borrows would be to try and convince ourselves that we have not just ruled out *one* counterexample for the desired optimization in `example1`, but *all possible* counterexamples. However, it turns out that this would be a doomed enterprise: the semantics is not yet quite right!

Specifically, if `example1` gets called with two pointer values *carrying the same tag*, then the dynamic analysis as described so far does not have any problem with both of them aliasing. It cannot

differentiate two references with the same tag. Such duplicate tags are possible because unsafe code can make copies of any data (*e.g.*, using `transmute_copy`), including mutable references.

The problem is that, when we just consider `example1`, the tags of `x` and `y` are provided by our caller and thus untrusted. To be able to reason based on tags and borrow stacks, we need to be sure that both references have unique tags that are not used by any other pointer value. (Unsafe code can *duplicate* tags but it cannot *forge* them—there is just no language operation for that.) This is achieved by inserting *retagging* instructions. **`retag`** is an administrative instruction that makes sure that references have a fresh tag. It is automatically inserted by the compiler—in particular, all arguments of reference type get retagged immediately when the function starts executing:

```
1  fn example1(x: &mut i32, y: &mut i32) -> i32 {
2    retag x; // equivalent to: `x = &mut *x;`
3    retag y; // equivalent to: `y = &mut *y;`
4    *x = 42;
5    *y = 13;
6    return *x; // We want to optimize this to return the constant 42.
7  }
```

As indicated in the comments, retagging achieves the desired effect by basically performing a reborrow. In the fragment of Rust we have considered so far, **`retag`** `x` behaves exactly like `x = &mut *x`, which means `x` still points to the same location, but we follow the usual Stacked Borrows steps for both using (the old value of) `x` and creating a new reference (USE-2, NEW-MUTABLE-REF).

Now we are finally ready to give a proof sketch for why performing the desired optimization in line 6 of this program is correct. If the program does not conform to the Stacked Borrows discipline, then there is nothing to show because the program is considered to have undefined behavior. So let us proceed under the assumption that the program does conform to Stacked Borrows:

(1) Let us say that after the **`retag`** on line 2, `x` has value Pointer(ℓ, t). We know that no other pointer value has tag t, and that this tag is at the top of ℓ's borrow stack.
(2) `x` is not used until line 4, so *if* any code in between has any effect on the value or the stack of ℓ, it will do that through a pointer value with a different tag. That tag must be below t in the stack, because we established in (1) that t is at the top. This means that using a pointer value with a different tag will pop t off the stack. However, for line 4 to pass the analysis, t must still be in the stack. Thus, ℓ could *not* have been accessed in between line 2 and line 4, and t must still be at the top of ℓ's stack. After executing line 4, the stack remains unchanged; but we now know that the value stored at ℓ is 42.
(3) Finally, in line 6, we can repeat the same argument again to show that *if* t is still in ℓ's stack, no access to ℓ could have occurred in the meantime. Hence, we can conclude that ℓ still contains value 42, and we can perform the desired optimization.

What is interesting about this argument is that we never argued explicitly about whether `x` and `y` are aliasing or not! As a result, this particular optimization can be generalized to an entire optimization *pattern*, where the access of `y` is replaced by *any* code that does not mention `x`:

```
1  fn example1(x: &mut i32, /* more arguments */) -> i32 {
2    retag x;
3    /* replace this line with any code not using x */
4    *x = 42;
5    /* replace this line with any code not using x */
6    return *x; // We want to optimize this to return the constant 42.
7  }
```

This pattern demonstrates that `x` is a "unique pointer": code not using `x` cannot possibly have an effect on the memory `x` points to.

In particular, the optimization also applies when the code not using `x` is a call to an *unknown function*. That is, Stacked Borrows allows us to do something that is unthinkable in the typical C/C++ compiler: we have a reference, `x`, that was passed in by the environment (so the pointer value is "escaped" in the sense that it is known to the environment), and still, we can call a foreign function `f()`, and *as long as we do not pass* `x` *as an argument to* `f`, we can assume that `f` neither reads from nor writes to `x`. Furthermore, we can make this assumption without doing any inlining, using only intraprocedural reasoning—a "holy grail" of alias analysis.

Where is **`retag`** *inserted?* The **`retag`** in the previous example was crucial because it allowed the compiler to assume that `x` actually had a unique tag. This optimization, as well as all of the ones we are going to see, can only be performed on references that have been retagged "locally" (within the function under analysis), because only then can we make the necessary assumptions about the tag of the reference and the borrow stack of the location it points to. So where exactly **`retag`** is inserted becomes an important knob in Stacked Borrows that determines for which references optimizations like the one above can be performed.

For now, we expect retagging to happen any time a reference gets passed in as an argument, returned from a function, or read from a pointer. Basically, any time a reference "enters" our scope, it should get retagged:

```
fn retag_demo(x: &mut i32, y: &&i32) {
  retag x; // All function arguments of reference type...
  retag y; // ...get retagged.
  let z = *y;
  retag z; // We also retag references read from a pointer...
  let incoming_ref = some_function_returning_a_ref();
  retag incoming_ref; // ...and references that functions returned to us.
}
```

3.5 Shared References

So far, we have only considered mutable references and raw pointers. We have seen that Stacked Borrows enforces a form of uniqueness for mutable references, enough to justify a program transformation to reorder a memory access around unknown code. In this section, we are going to look at shared references. The goal is to enforce that they are read-only, again to justify a program transformation that reorders a memory access around unknown code.

Just as we did for mutable references, we arrive at the stack principle for shared references by rephrasing what the borrow checker enforces about them in a way that avoids mentioning lifetimes: *every use of the reference (and everything derived from it) must occur before the next* mutating *use of the referent (after the reference got created), and moreover the reference must not be used for mutation.*

To see how this plays out, let us again consider a simple example involving references to integers:

```
let mut local = 42;
let x = &mut local;
let shared1 = &*x; // Derive two shared references...
let shared2 = &*x; // ...from the same mutable reference, x.
let val = *x; // Use all *three* references...
let val = *shared1; // ...interchangeably...
let val = *shared2; // ...for read accesses.
*x += 17; // Use x again for a write access.
let val = *shared1; // Error! shared1 used after x got mutated.
```

Despite the fact that x, shared1 and shared2 alias, this program is fine until line 8—but in line 9, the stack principle for shared references is violated: a use of the reference shared1 in line 9 occurs after a mutating use of the referent in line 8.

To model this in Stacked Borrows, we introduce another kind of item that can exist in the borrow stack:

$$\mathit{Item} \triangleq \mathsf{Unique}(t) \mid \mathsf{SharedRO}(t) \mid \mathsf{SharedRW}(t) \mid \ldots$$

The new item SharedRO(t) ("shared read-only") indicates that references tagged with t are allowed to read from but not write to the location associated with this stack. We also equip SharedRW with a tag. This means we can now speak about the "tag of an item", which will be useful in READ-1. In NEW-MUTABLE-RAW-1, we just push SharedRW($\bot$) instead of SharedRW (for now this is the only kind of SharedRW item).

We amend the existing Stacked Borrows rules as follows:

Rule (NEW-SHARED-REF-1)**.** Any time a new shared reference gets created (&expr) from some existing pointer value Pointer(ℓ, t), first of all this is considered a *read access* to that pointer value (so we follow READ-1 below). Then we pick some fresh tag t', use Pointer(ℓ, t') as the value for the shared reference, and add SharedRO(t') to the top of the stack for ℓ.

Rule (READ-1)**.** Any time a pointer with value Pointer(ℓ, t) gets *read from*, an item with tag t (*i.e.*, Unique(t), SharedRO(t) or SharedRW(t)) must exist in the stack for ℓ. Pop items off the stack until all the items above the item with tag t are SharedRO(_). If no such item exists in the stack, the program violates the stack principle. (This rule trumps the existing USE-2, which only gets used for writes now.)

Notice that we leave the rules for writing unchanged, which means that even if SharedRO(t) *is* in the stack, a reference tagged t cannot be used to write, as that requires a Unique(t) or SharedRW(t).

The key point in READ-1 (and the key difference from USE-2) is that reading with Pointer(ℓ, t) does *not* end up with an item with tag t being on top of the stack! There may be some SharedRO above it. This reflects the fact that two shared references can be used for reading without "disturbing" each other; they do not pop the other reference's item off the stack. In contrast, write accesses (which are still governed by USE-2) require that the item with the tag of the pointer used for the access becomes the top item on the stack.

Consequently, a key invariant that this system maintains is that if there are any SharedRO's in the stack, they are all adjacent at the top. Notice how all operations that would push another kind of item (creating a mutable reference or a raw pointer) count as a write access, so they would first make some Unique(_) or SharedRW(_) the top of the stack by popping off all SharedRO's above them. *Never is a* Unique(_) *or* SharedRW(_) *pushed on top of a* SharedRO(_).

With these rules, the example program executes as follows:

```
1  let mut local = 42; // Stored at location ℓ, with tag 0.
2  let x = &mut local; // = Pointer(ℓ, 1)
3  // Push tag of x on the stack: h(ℓ) = (42, [Unique(0), Unique(1)]). (NEW-MUTABLE-REF)
4  let shared1 = &*x; // = Pointer(ℓ, 2)
5  let shared2 = &*x; // = Pointer(ℓ, 3)
6  // New tags are pushed: h(ℓ) = (42, [Unique(0), Unique(1), SharedRO(2), SharedRO(3)]). (NEW-SHARED-REF-1)
7  let val = *x;
8  // Check: Unique(1) in the stack, all items above are SharedRO(_). (READ-1)
9  let val = *shared1;
10 // Check: SharedRO(2) in the stack, all items above are SharedRO(_). (READ-1)
11 let val = *shared2;
12 // Check: SharedRO(3) in the stack, all items above are SharedRO(_). (READ-1)
```

```
13  *x += 17;
14  // Pop until Unique(1) is at the top: h(ℓ) = (59, [Unique(0), Unique(1)]). (USE-2)
15  let val = *shared1;
16  // Analysis error! The tag of shared1 is not in the stack. (READ-1)
```

Observe how the rule for reading references allows `x`, `shared1`, and `shared2` to happily coexist (even the "XYXY" pattern is allowed), but the moment we write to `x`, the SharedRO(_) items get removed from the stack and the corresponding shared references may no longer be used.

3.6 An Optimization Exploiting Read-Only Shared References

To see how Stacked Borrows' treatment of shared references is helpful, let us consider a function that could benefit from an optimization exploiting that shared references are read-only:

```
1  fn example2(x: &i32, f: impl FnOnce(&i32)) -> i32 {
2    retag x;
3    let val = *x / 3;
4    f(x);
5    return *x / 3; // We want to optimize this to return val.
6  }
```

This time, we use a closure `f` to reflect the idea that arbitrary code can run in between line 3, where we read `x` for the first time, and line 5, where we want to optimize away the recomputation of `*x / 3`. Unlike in the case of mutable references, we even give that unknown code access to our reference `x`! It is a read-only reference, though, so `f` should not be able to write through it.

Again we can craft a counterexample that prohibits this optimization under a naive semantics:

```
1  fn main() {
2    let mut local = 6;
3    let x = &local;
4    let result = example2(x, |inner_x| {
5      retag inner_x;
6      let raw_pointer: *mut i32 = unsafe { mem::transmute(inner_x) };
7      unsafe { *raw_pointer = 15; }
8    });
9    println!("{}", result); // Prints "5" (aka 15/3).
10 }
```

Here we create a closure, for which the syntax in Rust is `|args| body`. The interesting part is on lines 6 and 7, which are the body of the closure `f` that we pass to `example2`. There, we circumvent the restriction that one cannot write to a shared reference, by calling `transmute`, which is Rust's unchecked cast operation. This lets us turn the (read-only) shared reference into a (writable) raw pointer—and then we write to it.

However, under Stacked Borrows, this program has undefined behavior (as we would hope). The `transmute` affects neither the tag t of the pointer nor the borrow stack. Thus, in line 7, when we follow USE-2, we fail to find a Unique(t) on the borrow stack. The only item with tag t is the SharedRO(t) that was added by the **`retag`** in line 5.

In more detail, here is the step-by-step execution of our counterexample for the optimization of `example2` (the closure makes the control flow a bit more complicated—the numbers in brackets indicate the order of execution):

```
1  fn main() {
2    let mut local = 6;  // Stored at location ℓ, with tag 0.
3    let x = &local;  // = Pointer(ℓ, 1)
4    // [0] h(ℓ) = (6, [Unique(0), SharedRO(1)]) (NEW-SHARED-REF-1)
```

```
5     // Next: jump to example2 (line 17).
6     let result = example2(x, |inner_x| {
7       retag inner_x;
8       // [3] inner_x = Pointer(ℓ, 3), h(ℓ) = (6, [. . . , SharedRO(1), SharedRO(2), SharedRO(3)])
9       let raw_pointer: *mut i32 = unsafe { mem::transmute(inner_x) };
10      // [4] raw_pointer = Pointer(ℓ, 3), h(ℓ) = (6, [. . . , SharedRO(1), SharedRO(2), SharedRO(3)])
11      unsafe { *raw_pointer = 15; }
12      // [5] Analysis error! No Unique(3) in the stack for write access. (USE-2)
13    });
14    println!("{}", result); // Prints "5" (aka 15/3).
15  }
16
17  fn example2(x: &i32, f: impl FnOnce(&i32)) -> i32 {
18    retag x;
19    // [1] x = Pointer(ℓ, 2), h(ℓ) = (6, [Unique(0), SharedRO(1), SharedRO(2)])
20    let val = *x / 3;
21    // [2] Check: SharedRO(2) is in the stack, all items above are SharedRO(_). (READ-1)
22    f(x); // Next: jump to closure body (line 6).
23    return *x / 3; // We want to optimize this to return val.
24  }
```

3.7 A Proof Sketch for the Optimization on Shared References

Again we have ruled out one particular counterexample to the desired optimization, but what we really need to do is to argue that the optimization is correct in any possible context. Here is the relevant code again:

```
1  fn example2(x: &i32, f: impl FnOnce(&i32)) -> i32 {
2    retag x;
3    let val = *x / 3;
4    f(x);
5    return *x / 3; // We want to optimize this to return val.
6  }
```

The argument now goes as follows:

(1) Assume that after the **retag** in line 2, x has value Pointer(ℓ, t). We know that SharedRO(t) is at the top of the borrow stack for ℓ. Let us call the current scalar value stored in that location s. The goal is to show that in line 5, ℓ *still* stores the value s.
(2) While f executes, we know that *any* write access will pop off *all* SharedRO's from the stack. This relies on the invariant that all SharedRO's in the stack are sitting on top. So we conclude that as long as SharedRO(t) is in the stack, the value stored at ℓ remains s.
(3) Finally, in line 5, we get to assume that SharedRO(t) is still in the stack, because otherwise the program would violate Stacked Borrows (there exists no other item with that tag). As a consequence, ℓ still stores s, which justifies the optimization.

4 STACKED BORROWS, PART II: MORE ADVANCED TRANSFORMATIONS

So far, we have seen two optimizations that exploit the assumption that even unsafe code has to conform to Stacked Borrows, a dynamic analysis that mirrors the static analysis performed by the Rust borrow checker. Both of these followed a similar pattern: some reference got retagged (step (1) in the proof sketches), then we made some change / observation (we wrote to the mutable reference / read the shared reference), then some unknown code got executed (2), and then we used our original reference again (3). The retag makes our reference the "topmost" reference in the stack,

and the final use asserts that it is still in the stack. This works well for reordering instructions *up* across unknown code, and in fact both transformations we considered so far can be reduced to such a reordering. For `example1`, we basically performed the following transformation (abstracting away the unknown code as functions `g` and `f`):

```
1  retag x;                          1  retag x;
2  g();                              2  g();
3  *x = 42;                          3  *x = 42;
4  f();                        ⇒     4  let x_val = *x;
5  let x_val = *x;                   5  f();
6  return x_val;                     6  return x_val;
```

This is "just" reordering the read `*x` up across the `f()`, swapping lines 4 and 5. Now the write to `x` and the read from it are directly adjacent in lines 3 and 4, so forwarding the 42 is a trivially correct transformation.

Similarly, `example2` "just" requires moving a read `*x` up across `f(x)`. Then we have two reads right next to each other; deduplicating those is easy to justify. The difference from `example1` is that this is a read of a *shared* reference and that the unknown code in `f` actually has access to `x`.

However, in some cases it is also interesting to be able to move a memory access *down* across unknown code. This is harder, because it *extends* the liveness range of a reference! For example, consider the following function:

```
1  fn example2_down(x: &i32, f: impl FnOnce(&i32)) -> i32 {
2    retag x;
3    let val = *x;
4    f(x);
5    return val; // Can return *x instead.
6  }
```

We might want to move the read of `x` down from line 3 to line 5, across the call to `f`. This reduces register pressure because we do not need to remember `val` around the call to `f`.

For mutable references, we can similarly move reads down across code that does not need them, but the more interesting case is moving a *write* access down:

```
1  fn example3_down(x: &mut i32, f: impl FnOnce()) {
2    retag x;
3    *x = 42; // This write is redundant -- we would like to optimize it away.
4    f();
5    *x = 13;
6  }
```

Removing the redundant write here boils down to moving the write to `x` *down* from line 3 to line 5. Then we end up with two adjacent writes to the same reference, and the first one can be removed. However, this means that unlike in any prior optimization we considered (and in fact unlike in the vast majority of optimizations that are typically performed), `f` will be called with a different value stored in `x` than in the original program! We will see later why this can work.

4.1 Protectors

It turns out that Stacked Borrows, as we have presented it so far, does not permit these optimizations that move operations down. Here is a counterexample for `example2_down`:

```
1  fn main() {
2    let mut local = 42;  // Stored at location ℓ, with tag 0.
3    let raw_pointer = &mut local as *mut i32;
```

```
4     let val = example2_down(
5       unsafe { &*raw_pointer }, // = Pointer(ℓ, 2)
6       |x_inner| unsafe { retag x_inner; *raw_pointer = 17; }, // Changes *x.
7     );
8     println!("{}", val); // Prints 42.
9   }
```

The closure that we pass as `f` changes the value stored in `x` by writing to `raw_pointer`, an alias of `x`. This is allowed by Stacked Borrows: immediately before the write, the borrow stack for ℓ looks like [Unique(0), Unique(1), SharedRW($\bot$), SharedRO(2), SharedRO(3)], where 1 is the tag of the temporary mutable reference created in line 3, 2 is the tag of the shared reference created in line 5, and 3 is the tag of `x` inside `example2_down` (retagging assigns a new tag there). In this situation, we are allowed to perform a write with a raw pointer thanks to the SharedRW($\bot$) in the stack, and the resulting stack is [Unique(0), Unique(1), SharedRW($\bot$)]. This removes the tag of `x` from the stack, but since `x` does not get used again, that is not a violation of the rules we have set so far.

However, in Rust, a reference passed to a function must *outlive* the function call—*i.e.*, its lifetime must last at least for the duration of the call. Intuitively, since the lifetime of a reference relates to how long the item for that reference's tag is in the stack, the problem in the above counterexample is that the lifetime of `x` ends while `example2_down` is still running, when SharedRO(2) gets popped off the stack. To enable the desired optimization, we thus want to treat the counterexample as incurring undefined behavior, by reflecting Rust's "outlives" rule into Stacked Borrows.

To prevent SharedRO(2) from being popped off while `example2_down` still runs, we introduce the notion of a *protector*: an item in the stack can be *protected by a function call*, which means that while that function call is still ongoing, if the item gets popped off the stack, that is a Stacked Borrows violation and hence undefined behavior.

Formally, we extend the items in the borrow stack with an optional *call ID*:

$$\mathit{CallId} \triangleq \mathbb{N} \qquad \mathit{Item} \triangleq \mathsf{Unique}(t, c) \mid \mathsf{SharedRO}(t, c) \mid \mathsf{SharedRW}(t, c)$$

Here, c is an element of $\mathit{CallId}^? \triangleq \mathit{CallId} \cup \{\bot\}$. Every call ID represents a function call (we can imagine it being associated with the stack frame). We also assume that the semantics keeps track of the set of call IDs that correspond to function calls that have not returned yet. We use $\mathsf{Unique}(t)$ as notation for $\mathsf{Unique}(t, \bot)$, and similarly for the other kinds of items.

So, when do newly pushed items get protectors? Remember that we have **`retag`** operations every time a reference is passed in as an argument, read via a pointer, or returned from another function. The idea is that we make the first case, the **`retag`** of the arguments, special in the sense that this retag will protect any new items that it adds with the call ID of the current function call. This ensures that such items will not be popped during said function call. Syntactically, we will write **`retag[fn]`** `x` to indicate that `x` is being retagged in the prologue of a function, and thus its items get protected.

Now we add the following to our rules for Stacked Borrows:

Rule (RETAG-FN)**.** When pushing new items to a location's borrow stack as part of a **`retag[fn]`** (in NEW-MUTABLE-REF or NEW-SHARED-REF-1), these items have their protector set to the call ID of the current function call.

Rule (PROTECTOR)**.** Any time an item gets popped (by USE-2 or READ-1), check if it has a protector ($c \neq \bot$). If it does, and if that call ID corresponds to a function call that is still ongoing (that is, the corresponding stack frame is still in the call stack), we say that the protector is "active", and popping an active protector is undefined behavior.

The code of `example2_down` changes a bit, to reflect that we need the new kind of retag for `x`:

```
1  fn example2_down(x: &i32, f: impl FnOnce(&i32)) -> i32 {
2    retag[fn] x;
3    let val = *x;
4    f(x);
5    return val; // Can return *x instead.
6  }
```

If we now consider the previous counterexample again, this time the borrow stack of ℓ at the beginning of the execution of the closure is:

$$[\text{Unique}(0), \text{Unique}(1), \text{SharedRW}(\bot), \text{SharedRO}(2), \text{SharedRO}(3, c)]$$

Here, c is the call ID of the call to `example2_down`. But this means when the closure executes and performs its raw pointer write, it will hit PROTECTOR: a raw pointer write has to make $\text{SharedRW}(\bot)$ the top of the stack, which requires popping off $\text{SharedRO}(3, c)$, but that is not allowed because the protector c is still active.

4.2 Proof Sketches for the Optimizations

We have seen that protectors successfully rule out what used to be a counterexample against optimizing `example2_down`. In fact, protectors enable us to validate the transformation:

(1) Assume that after the **`retag[fn]`** in line 2, `x` has value $\text{Pointer}(\ell, t)$. We know that the top item of the borrow stack for location ℓ is $\text{SharedRO}(t, c)$. Here, c is the call ID of `example2_down`. We remember the current scalar value stored at ℓ as s.
(2) While `f` executes, we know that any write access to ℓ would pop all $\text{SharedRO}(\_, \_)$ off the stack. That would immediately be a Stacked Borrows violation because $\text{SharedRO}(t, c)$ has an active protector c. Thus no write access can happen.
(3) As a consequence, when `f` returns, ℓ still stores s. This justifies the transformation.

Almost the same argument works for `example3_down`, which moves a *write* down across a function call. Again, `x`'s tag is at the top of the borrow stack when `f` gets called, and it is protected. Any attempt by `f` to access `x` (reading or writing) will pop off that item, which is not allowed due to the protector. Thus `f` cannot observe the value stored in `x`, and we are free to do the write later.[4] What is remarkable about this transformation is that the compiler ends up calling `f` with *a different value stored in* `x`, but we can show that this cannot affect the behavior of programs complying with Stacked Borrows, even if `f` runs into an infinite loop and we never get to step (3).

5 SUPPORTING INTERIOR MUTABILITY

We have seen that Stacked Borrows supports all of the transformations that it was designed to enable: moving uses of shared and mutable references up and down across unknown code. This already gives the compiler the necessary freedom to exploit reference types for its alias analysis. But to have Stacked Borrows adopted, we additionally need to make sure that the large body of existing Rust code is actually compatible with the rules of Stacked Borrows. Unfortunately, there is one feature of Rust that is in direct contradiction to our treatment of shared references: *interior mutability*. Contrary to the exclusion principle introduced in §2, Rust in fact *does* permit mutation of shared references under some controlled circumstances. For example, Rust provides a type called `Cell` that permits mutation through a `set` method that works with a shared reference:

[4]Here we assumed that `f` cannot return in other ways, such as through an exception/unwinding. Unlike our formal model, Rust supports unwinding. The same transformation is still possible, the write just has to be pushed down into both the normal continuation and the unwinding continuation.

```
1  fn cells(x: &Cell<i32>, y: &Cell<i32>) -> i32 {
2    x.set(13);
3    y.set(42);
4    return x.get(); // Can return 13 or 42, depending on whether they alias.
5  }
```

A `&Cell<i32>` is essentially a safe version of an `int*` in C: it can be read from and written to by many parties at any time.[5] It may appear that giving safe code access to a type like `Cell` would break the entire safety promise of Rust, which after all rests on the exclusion principle—but `Cell` is carefully restricted to prevent this. Namely, `Cell` only allows copying data into and out of the `Cell`; it is not possible to obtain a reference *into* a `Cell`. Essentially, `Cell` rules out interior pointers.[6] Without interior pointers, most of the hard problems around shared mutation disappear, and the remaining issues are handled by the fact that `&Cell<i32>` is still a reference with a lifetime so that we can ensure we do not use this reference when it is no longer valid.

Of course, allowing mutation of aliased data makes alias analysis as hard as it is in C. However, the Rust designers were aware that interior mutability would wreak havoc with the otherwise very strong aliasing guarantees that their type system provides, so they decided early on that the programmer must "mark" data that will be subject to interior mutability, in a way that can be recognized by the compiler. To this end, the standard library contains the "magic" type `UnsafeCell<T>`. An `UnsafeCell<T>` is basically the same as a `T`; however, the type is specially recognized by the compiler. This is exploited in the rule for mutation of shared data, which says that shared references may not be used to perform mutation *except* when the mutation happens "inside" an `UnsafeCell`. For example, `Cell<T>` is a newtype built around `UnsafeCell<T>` with a safe API.

We exploit this information in Stacked Borrows and exempt the memory inside an `UnsafeCell<T>` from the usual read-only restriction for shared references. To track references into memory inside an `UnsafeCell<T>`, we make use of the SharedRW items that we have already seen for raw pointers: raw pointers and interior mutable shared references both permit mutable aliasing, so it makes sense to treat them similarly.

"Partial" interior mutability. What makes this complicated is that, in general, a shared reference can have "partial" interior mutability. So far, we have pretended that a reference points to a single location and affects a single stack, but of course in reality a reference like `&i32` spans 4 locations. All the rules that we have seen (both for creating references/raw pointers and for extra actions to be performed on memory accesses) apply to the stack of *every location* that is covered by the reference (with the size determined by the type `T`). However, with `&(i32, Cell<i32>)`, the first 4 bytes are *not* inside an `UnsafeCell` and are thus subject to the full set of Stacked Borrows rules for shared references, but the second 4 bytes *are* inside an `UnsafeCell`, and Stacked Borrows will permit mutation of shared data for those 4 bytes only.

Consequently, when creating a shared reference with "partial" interior mutability, we *cannot* perform the same actions for all locations that the reference covers. We first pick a new tag t, and then we must find the `UnsafeCell`s in the memory region that the reference points to. This is basically a type-based traversal of the memory the reference covers. For all locations outside of the `UnsafeCell`s, we proceed as before and push a new *read-only* SharedRO(t, c) on top of the stack, after following READ-1 (remember that creating a shared reference counts as a read access). For all locations *inside* an `UnsafeCell`, we instead add to the stack the *read-write* item SharedRW(t, c), which gives us the same aliasable mutation that mutable raw pointers enjoy.

[5] In fact, after inlining, the function `cells` above looks just like C code writing to and reading from `int*`.
[6] It also rules out data races by employing the type system to ensure that `&Cell<T>` cannot be moved to another thread. For further discussion of the `Send` trait that enables this restriction, see Jung et al. [2018].

Creating a reference is not always an access. The other difference from what we have seen so far is that, for the part inside the `UnsafeCell<T>`, creating a shared reference does *not* count as a read access, and the new item does *not* get added at the top of the stack (but in the middle instead).

To explain why we cannot treat creating a shared reference as a read access when interior mutability is involved, we have to briefly discuss `RefCell<T>`. `RefCell<T>` is a Rust standard library type that is somewhat similar to `Cell<T>`, but it *does* allow interior pointers. It achieves safety by tracking at run-time how many mutable and shared references to the data exist, and making sure that there always is either exactly one mutable reference, or an arbitrary number of (read-only) shared references. However, this means that it is possible to call the following function from safe code *in a way that the two references* `shared` *and* `mutable` *alias*:

```
1  fn evil_ref_cell(shared: &RefCell<i32>, mutable: &mut i32) {
2    retag[fn] shared; retag[fn] mutable;
3    let more_shared = &*shared;
4    *mutable = 23;
5  }
```

If creating a shared reference (with the reborrow in line 3, and in fact already for the **`retag`** in line 2) counted as a read access, that would violate uniqueness of `mutable`! After all, one important property of mutable references is that there are no other accesses (read or write) to that memory while the reference is used. We also cannot add a new item for `more_shared` on top of the stack; that would violate the invariant that the tag of `mutable` is at the top of the stacks it points to (after retagging)—an invariant that was crucial in our proof sketches.

Consequently, we need to create a shared reference without popping things off the stack. However, we still want the item to be "next to" the one it is derived from. So, when creating a new shared reference from Pointer(ℓ, t), we add the new SharedRW just above where t is in the stack.

Rule (NEW-SHARED-REF-2)**.** When creating a new shared reference from some existing pointer value Pointer(ℓ, t), we pick some fresh tag t' and use Pointer(ℓ, t') as the value for the shared reference.

For all locations covered by this reference that are inside an `UnsafeCell`, we find the item for t in the borrow stack, and we add SharedRW(t') just above that item. (If there is no such item, the program has undefined behavior.) For the remaining locations, this counts as a read access with the old tag t (see READ-1). Then we push SharedRO(t') to the top of the stack.

To keep the rules for shared references with interior mutability and mutable raw pointers consistent (both use SharedRW, after all), we adjust the latter to not count as an access either:

Rule (NEW-MUTABLE-RAW-2)**.** When a mutable raw pointer gets created via a cast (`expr as *mut T`) from some mutable reference (`&mut T`) with value Pointer(ℓ, t), we find the item for t in the borrow stack, and we add SharedRW($\bot$) just above that item. (If there is no such item, the program has undefined behavior.) Then the pointer gets value Pointer($\ell, \bot$).

With these rules, reborrowing `shared` will just add some SharedRW in the middle of the stack—so even if `mutable` aliases, we maintain the property that the *top item* of the stack is a Unique with the same tag as `mutable`. That is enough to keep our proofs working, and it also permits a program calling `evil_ref_cell` with aliasing pointers.

This means, however, that we can end up with many SharedRW next to each other in the stack, *e.g.*, when creating a bunch of shared references with interior mutability from the same mutable reference. Just as we considered the adjacent SharedRO on top of the stack as one "group" of items that can all be used without removing each other from the stack, we want to do the same with such a "group" of adjacent SharedRW items. Thus we need to adjust the rules for (read and write)

accesses in such a way that if any of an adjacent group of SharedRW is used, the others in that group all remain in the stack.

For writes, this looks as follows (also incorporating PROTECTOR):

Rule (WRITE-1)**.** On any write access using pointer value Pointer(_, t), do the following for each location ℓ affected by the access: pop the stack until either the top item is Unique(t, _), or a SharedRW(t, _) exists in the top "group" of adjacent SharedRW in the stack (*i.e.*, SharedRW(t, _) is in the stack, and there are only SharedRW above it). If this is not possible, or if this pops off an item with an active protector, the program has undefined behavior.

Read accesses and disabled items. Read accesses, too, need to account for the fact that we now have *tagged* SharedRW items. We want to be able to read from a SharedRW item without invalidating the other SharedRW or SharedRO items that may be adjacent to it. (This parallels what we already did for reading SharedRO items in READ-1 and writing SharedRW items in WRITE-1.)

However, we still need to make sure that there are never any Unique items left above the item that justifies the access, and there is a slight twist to how we do this. For reasons we will explain shortly, instead of popping items off the stack, we keep all the items where they are but mark the Unique ones as "disabled". This means in particular that reading from a SharedRW will maintain validity of *all* SharedRW items above it on the stack (not just the ones that are adjacent to it).

To this end, we introduce a new kind of item:

$$\mathit{Item} \triangleq \mathsf{Unique}(t, c) \mid \mathsf{SharedRO}(t, c) \mid \mathsf{SharedRW}(t, c) \mid \mathsf{Disabled}$$

And we change the read rule as follows:

Rule (READ-2)**.** On any read access using pointer value Pointer(_, t), do the following for each location ℓ affected by the access: find the topmost (non-Disabled) item with tag t in the stack (there can be several if $t = \bot$). Replace all the Unique(_, _) above it by Disabled. If any of these Unique(_, _) has an active protector, the program has undefined behavior.

The reason for this quirk in READ-2 is that testing with some real Rust code quickly revealed the following pattern. This pattern is widely used (and thus should *not* induce undefined behavior), but it *would* induce undefined behavior without the "disabled" approach:

```
1  fn make_raw(y: &mut i32) -> *mut i32 { retag[fn] y; y as *mut i32 }
2
3  fn bad_pattern(x: &mut i32) {
4    retag[fn] x; // Assume x = Pointer(ℓ, t_x).
5    let raw_ptr = make_raw(x); // Desugars to `make_raw(&mut *x)`.
6    // h(ℓ) = [..., Unique(t_x), Unique(t_tmp), Unique(t_y), SharedRW(⊥)]
7    let val1 = *x;
8    // h(ℓ) = [..., Unique(t_x)]
9    let val2 = unsafe { *raw_ptr }; // Fails because SharedRW(⊥) is no longer in the stack!
10 }
```

Right before line 7, the borrow stack for ℓ contains items for the intermediate mutable references that were created "between" `x` and `raw_ptr`: the temporary that gets introduced because passing a reference to a function in Rust is implicitly reborrowing, and the new tag that gets picked when `y` gets retagged in `make_raw`.

The trouble is that reading from `x` has to invalidate the Unique items above it: the `example3_down` transformation to move a write down across a function call relies on the fact that other code cannot read from memory covered by a mutable reference without causing a violation. But if we are forced to adhere to the stack discipline, we can only invalidate the Unique by first popping off the SharedRW($\bot$), which makes line 9 illegal because SharedRW($\bot$) is no longer in the stack.

To fix this[7], we deviate from the stack discipline in the way we handle removal of tags on a read access. Namely, instead of popping items off the stack until there are no more Unique above the accessed item (which would pop off some SharedRW as well), we just *disable* the Unique that are above the accessed item while leaving the SharedRW items alone. That way, the SharedRW items do not suffer any "collateral damage" from the popping of Unique items, and the above programming pattern is permitted.

This is the last adjustment we need. Let us now take a step back and consider the semantics in its entirety—formally.

6 FORMAL DEFINITION OF THE OPERATIONAL SEMANTICS

The purpose of this section is to formally define what we have so far described informally in this paper. The full formal definition can be found in our technical appendix [Jung et al. 2019].

6.1 High-Level Structure

At a high level, we are defining Stacked Borrows as a labeled transition system where the labels are *events* (the most important ones being read and write accesses and retagging) and the state is described by the following record (some of the remaining relevant domains are shown in Figure 2):

$$\varsigma \in \mathit{SState} \triangleq \left\{ \begin{array}{rl} \textsc{stacks} : & \mathit{Stacks}, \\ \textsc{calls} : & \mathit{List}(\mathit{CallId}), \\ \textsc{nextptr} : & \mathit{PtrId}, \\ \textsc{nextcall} : & \mathit{CallId} \end{array} \right\} \qquad \xi \in \mathit{Stacks} \triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Stack}$$

Here, STACKS tracks the stack for each location, CALLS tracks the current list of active call IDs (needed to implement PROTECTOR), and NEXTPTR and NEXTCALL are used to generate fresh pointer IDs and call IDs.

This approach decouples Stacked Borrows from the rest of the language: the operational semantics of the language can just include an *SState* and take appropriate transitions on that state whenever a relevant event occurs. Therefore, unlike in the previous section, we do not use a single heap $h \in \mathit{Mem}$ for both the value and stack at each location. Only the stack is relevant for Stacked Borrows, and it is tracked by *SState* in its STACKS field; the actual values stored in the heap are assumed to be handled by the rest of the semantics. In Figure 2, we also re-define *Item* as a triple so that the common structure (a permission, a tag, and an optional protector) is reflected in the data.

The possible events and the most important transitions are likewise defined in Figure 2. These events include all memory accesses (reads, writes, allocations, and deallocations), as well as initiating and ending a function call (which is relevant for the tracking of call IDs to make protectors work) and, of course, retagging. Most of the parameters in these events are "input parameters", in the sense that they represent information that comes from the outside world into the Stacked Borrows subsystem. Only the call ID c in the call events and the new tag t_{new} in retag events are "output parameters", representing information that is returned by Stacked Borrows to the outside world.

The effect of every event on the Stacked Borrows state ς is described by a function that *computes* the next state given the previous one. We use ς **with** [FIELD := e] as notation for updating a single field of a record (and, as we will see later, also for updating elements of a finite partial map). Considering the very algorithmic nature of Stacked Borrows, we feel that it lends itself more to this computational style than a relational one.

[7]Note that in this example we could replace the dereference `*raw_ptr` in line 9 with `*x` because `make_raw` is the identity function. However, in general `make_raw` will actually do some work that we do not want to re-do.

Domains:

$$PtrId \triangleq \mathbb{N} \qquad \iota \in Item \triangleq Permission \times Tag \times CallId^?$$
$$t \in Tag \triangleq PtrId^? \qquad p \in Permission \triangleq \text{Unique} \mid \text{SharedRW} \mid \text{SharedRO} \mid \text{Disabled}$$
$$c \in CallId \triangleq \mathbb{N} \qquad S \in Stack \triangleq List(Item)$$

Events:

$$AccessType \triangleq \text{AccessRead} \mid \text{AccessWrite} \qquad m \in Mutability \triangleq \text{Mutable} \mid \text{Immutable}$$
$$RetagKind \triangleq \text{Default} \mid \text{Raw} \mid \text{FnEntry} \qquad PtrKind \triangleq \text{Ref}(m) \mid \text{Raw}(m) \mid \text{Box}$$
$$\begin{aligned} \varepsilon \in Event \triangleq\ & \text{EAccess}(a, \text{Pointer}(\ell, t), \tau) \quad \text{where } a \in AccessType, \tau \in Type \\ \mid\ & \text{ERetag}(\text{Pointer}(\ell, t_{old}), t_{new}, \tau, k, k') \quad \text{where } k \in PtrKind, k' \in RetagKind \\ \mid\ & \text{EAlloc}(\text{Pointer}(\ell, t), \tau) \mid \text{EDealloc}(\text{Pointer}(\ell, t), \tau) \\ \mid\ & \text{EInitCall}(c) \mid \text{EEndCall}(c) \quad \text{where } c \in CallId \end{aligned}$$

Selected transitions:

OS-ACCESS

$$\frac{\text{MemAccessed}(\varsigma.\text{STACKS}, \varsigma.\text{CALLS}, a, \text{Pointer}(\ell, t), |\tau|) = \xi' \qquad \varsigma' = \varsigma \textbf{ with } [\text{STACKS} := \xi']}{\varsigma \xrightarrow{\text{EAccess}(a, \text{Pointer}(\ell, t), \tau)} \varsigma'}$$

OS-RETAG

$$\frac{\begin{array}{c}\text{Retag}(\varsigma.\text{STACKS}, \varsigma.\text{NEXTPTR}, \varsigma.\text{CALLS}, \text{Pointer}(\ell, t_{old}), \tau, k, k') = (t_{new}, \xi', n') \\ \varsigma' = \varsigma \textbf{ with } [\text{STACKS} := \xi', \text{NEXTPTR} := n']\end{array}}{\varsigma \xrightarrow{\text{ERetag}(\text{Pointer}(\ell, t_{old}), t_{new}, \tau, k, k')} \varsigma'}$$

Fig. 2. Stacked Borrows domains and transitions.

In the following, we discuss the formal definitions of the most interesting events, which have been covered only informally so far: accesses (reads and writes), and retagging.

6.2 Memory Accesses

In Figure 3, we formalize READ-2 and WRITE-1 by implementing MemAccessed, the function defining what happens on EAccess events. The definition of the function itself is simple: it just iterates over all locations affected by the access and calls the helper function Access to compute their new stacks. To determine the set of affected locations, we need to know the size of the access; this is computed based on the type τ which is part of the event.[8] Access is a partial function: $\bot$ is used as return value to express that the access is illegal. If that is the case, our **with** operator propagates the failure, so MemAccessed also returns $\bot$ and the transition system (Figure 2) gets stuck.

For both reads and writes, Access (Figure 3) starts by finding the "granting item", defined by the Grants function. Grants(p, a) determines if an item with given permission p may be used to justify a memory access, with a indicating whether this is a read or a write access. For a write, only Unique and SharedRW are allowed (as in WRITE-1); for a read we also accept SharedRO (excluding only Disabled, as in READ-2). Access uses FindGranting to search the stack S top-to-bottom to find the topmost item which matches the given tag t and grants the current access. Implicitly, in the

[8]Types reflect just as much of the Rust type system as is needed for Stacked Borrows; see our appendix for the definition.

(* Defines whether p can be used to justify accesses of type a. *)
$\mathsf{Grants}(p : \mathit{Permission}, a : \mathit{AccessType}) : \mathbb{B} \triangleq p \neq \mathsf{Disabled} \land (a = \mathsf{AccessRead} \lor p \neq \mathsf{SharedRO})$

(* Finds the topmost item in S that grants access a to t.
Returns the index in the stack (0 = bottom) and the permission of the granting item. *)
$\mathsf{FindGranting}(S : \mathit{Stack}, a : \mathit{AccessType}, t : \mathit{Tag}^?) : (\mathbb{N} \times \mathit{Permission})^?$

$\mathsf{FindGranting}(S +\!\!+ [\iota], a, t) \triangleq \textbf{if } (\iota.\textsc{tag} = t \land \mathsf{Grants}(\iota.\textsc{perm}, a))$
$\textbf{then } (|S|, \iota.\textsc{perm}) \textbf{ else } \mathsf{FindGranting}(S, a, t)$

(* Finds the bottommost item above i that is incompatible with a write justified by p. *)
$\mathsf{FindFirstWIncompat}(S : \mathit{Stack}, i : \mathbb{N}, p : \mathit{Permission}) : \mathbb{N}^?$

(* Writes to Unique are incompatible with everything above. *)
$\mathsf{FindFirstWIncompat}(S, i, \mathsf{Unique}) \triangleq i + 1$

(* Writes to SharedRW are *compatible* with adjacent SharedRW, and nothing else.
So if the next item up is SharedRW then go on searching, otherwise stop. *)
$\mathsf{FindFirstWIncompat}(S, i, \mathsf{SharedRW}) \triangleq \textbf{if } i + 1 < |S| \land S(i+1).\textsc{perm} = \mathsf{SharedRW}$
$\textbf{then } \mathsf{FindFirstWIncompat}(S, i+1, \mathsf{SharedRW}) \textbf{ else } i + 1$

(* Computes the new stack after an access of type a with tag t.
Also depends on the active calls tracked by C. *)
$\mathsf{Access}(a : \mathit{AccessType}, S : \mathit{Stack}, t : \mathit{Tag}^?, C : \mathit{List}(\mathit{CallId})) : \mathit{Stack}^?$

$\mathsf{Access}(\mathsf{AccessRead}, S, t, C) \triangleq$
$\textbf{bind } (i, \_) = \mathsf{FindGranting}(S, \mathsf{AccessRead}, t) \textbf{ in}$
(* Disable all Unique above i; error out if any of them is protected. *)
$\textbf{if } \{S(j).\textsc{protector} \mid j \in (i, |S|) \land S(j).\textsc{perm} = \mathsf{Unique}\} \cap C = \emptyset$
$\textbf{then } S \textbf{ with}_{\,j \in (i,|S|) \land S(j).\textsc{perm} = \mathsf{Unique}} [j := (S(j) \textbf{ with } [\textsc{perm} := \mathsf{Disabled}])] \textbf{ else } \bot$

$\mathsf{Access}(\mathsf{AccessWrite}, S, t, C) \triangleq$
$\textbf{bind } (i, p) = \mathsf{FindGranting}(S, \mathsf{AccessWrite}, t) \textbf{ in}$
$\textbf{bind } j = \mathsf{FindFirstWIncompat}(S, i, p) \textbf{ in}$
(* Remove items at j and above; error out if any of them is protected. *)
$\textbf{if } \{S(i').\textsc{protector} \mid i' \in [j, |S|)\} \cap C = \emptyset$
$\textbf{then } S|_{[0,j)} \textbf{ else } \bot$

(* Read and write accesses are just lifted pointwise for each location. *)
$\mathsf{MemAccessed}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), a, \mathsf{Pointer}(\ell, t), n : \mathbb{N}) : \mathit{Stacks}^? \triangleq$
$\xi \textbf{ with}_{\,\ell' \in [\ell, \ell+n)} [\ell' := \mathsf{Access}(a, \xi(\ell'), t, C)]$

Fig. 3. Stacked Borrows access semantics.

base case of the recursion where the list is empty, FindGranting returns $\bot$ (as none of its patterns match). This only happens when there is no item that grants the desired access.

In Access, we use **bind** to denote the bind operation of the partiality monad: if FindGranting returns $\bot$, then that is propagated by Access; otherwise we proceed with the remaining computation.

```
(* Computes the new stack after inserting new item ι_new derived from old tag t_old.
   Also depends on the list of active calls C (used by Access). *)
GrantTag(S : Stack, t_old : Tag?, ι_new : Item, C : List(CallId)) : Stack? ≜
  (* Determine the "access" this operation corresponds to. Step (3). *)
  let a = (if Grants(ι_new.PERM, AccessWrite) then AccessWrite else AccessRead) in
  (* Find the item matching the old tag. Step (4). *)
  bind (i, p) = FindGranting(S, a, t_old) in
  if ι_new.PERM = SharedRW then
    (* A SharedRW just gets inserted next to the granting item. Step (5). *)
    bind j = FindFirstWIncompat(S, i, p) in InsertAt(S, ι_new, j)
  else
    (* Otherwise, perform the effects of an access and add item at the top. Step (5). *)
    bind S' = Access(a, S, t_old, C) in InsertAt(S', ι_new, |S'|)

(* Reborrow the memory pointed to by Pointer(ℓ, t_old) for pointer kind k and pointee type τ.
   prot indicates if the item should be protected. *)
Reborrow(ξ : Stacks, C : List(CallId), Pointer(ℓ, t_old), τ : Type, k : PtrKind, t_new : Tag?, prot : CallId?)
    : Stacks? ≜
  (* For each location, determine the new permission and add a corresponding item. *)
  ξ with (ℓ',fr)∈FrozenIter(ℓ,τ)[ ℓ' :=
      let p_new = NewPerm(k, fr) in (* Step (1). *)
      let ι_new = (p_new, t_new, prot) in (* Step (2). *)
      GrantTag(ξ(ℓ'), t_old, ι_new, C) ]
```

Fig. 4. Stacked Borrows retagging semantics (excerpt).

Having determined the granting item, Access proceeds to implement the rest of READ-2 and WRITE-1, respectively, as indicated by the comments in the definition.

6.3 Retagging

In §3-§5, Stacked Borrows took some extra actions any time a reference was created (NEW-MUTABLE-REF and NEW-SHARED-REF-2) or cast to a raw pointer (NEW-MUTABLE-RAW-2). We then explained `retag` as basically sugar for one of these operations. It turns out that we can simplify implementation and formalization by swapping things around: an assignment like `let x = &mut expr` in Rust becomes `let x = &mut expr; retag x` in our language with explicit retagging, and `retag` takes care of assigning a new tag (following NEW-MUTABLE-REF, in this case). The `&mut expr` itself just keeps the old tag. This is also the case for creating shared references and casting raw pointers. For the latter, we insert a `retag[raw]` to let retagging know that this is part of a reference-to-raw-pointer cast. Altogether, this means that accesses and `retag` cover everything we discussed in the paper, without the need to instrument reference creation or casts.

So, let us take a closer look at retagging. Or rather, let us first look at its core helper function, *reborrowing*, defined as Reborrow in Figure 4. Reborrowing takes as parameters the tag t_{old} and location ℓ of the *old* pointer (remember that retagging always starts with an existing pointer and updates its tag), the type τ the pointer points to, the tag t_{new} and pointer kind k of the *new* pointer, and *prot*, an optional call ID the new items will be protected by. The pointer kind k here determines if the new pointer is a reference, raw pointer or `Box`; this affects which new permission p_{new} will be used for the items we are about to add.

Reborrowing proceeds as follows for each location ℓ' covered by the pointer (that list of locations is computed by FrozenIter(ℓ, τ)):

(1) Compute the permission p_{new} we are going to grant to t_{new} for ℓ' (done by NewPerm[9]):
   - If `x` is a mutable reference (`&mut T`), $p_{new} \triangleq$ Unique.
   - If `x` is a mutable raw pointer (`*mut T`), $p_{new} \triangleq$ SharedRW.
   - If `x` is a shared reference (`&T`) or a constant raw pointer (`*const T`)[10], we check if ℓ' is *frozen* (definitely outside an `UnsafeCell`) or not. This is determined by FrozenIter based on the type τ. For frozen locations we use $p_{new} \triangleq$ SharedRO; otherwise $p_{new} \triangleq$ SharedRW.

(2) The new item we want to add is $\iota_{new} \triangleq (p_{new}, t_{new}, prot)$. Proceed in GrantTag to actually add that item to the stack.

(3) The "access" a that this operation corresponds to is a write access if p_{new} permits writing, and a read access otherwise.

(4) Find the granting item for this access with tag t_{old} in $\varsigma.\text{STACKS}(\ell')$.

(5) Check if $p_{new} =$ SharedRW.
   - If yes, add the new item ι_{new} just above the granting item.
   - Otherwise, perform the effects of a read or write access (as determined by a) to ℓ' with t_{old}. Then, push the new item ι_{new} on top of the stack.

Retagging now just needs to determine the parameters for reborrowing. Given the pointer kind k and the kind of retag k' (regular **`retag`**, **`retag[fn]`**, or **`retag[raw]`**), the new tag t_{new} and protector *prot* are computed as follows: raw pointers get new tag $\perp$ without a protector, but only on **`retag[raw]`** (otherwise they are ignored); `Box` pointers get a fresh tag but no protector; references get a fresh tag and on **`retag[fn]`** also a protector.

The full presentation of all formal details, including the event types we did not discuss, can be found in our technical appendix [Jung et al. 2019].

7 EVALUATION

As described in the introduction, we have evaluated Stacked Borrows in two ways. First, to ensure that the model actually accepts enough real Rust code to be a realistic option, we implemented this model in an existing Rust interpreter called *Miri*.[11] Secondly, we formalized the informal proof sketches given in the paper in Coq.

7.1 Miri

We implemented Stacked Borrows in Miri to be able to test existing bodies of **`unsafe`** Rust code and make sure the model we propose is not completely in contradiction with how real Rust code gets written. Moreover, this also served to test a large body of *safe* code (including code that

[9] We do not show all the helper functions here due to space limitations; they are given in full in our technical appendix.
[10] Rust actually has two kinds of raw pointers, mutable and constant. Constant raw pointers correspond to **`const`** pointers in C. We have not discussed constant raw pointers in this paper to keep the discussion more focused. They behave basically like untagged shared references.
[11] Available online at https://github.com/rust-lang/miri/.

relies on non-lexical lifetimes), empirically verifying that Stacked Borrows is indeed a dynamic version of the borrow checker and accepts strictly more code than its static counterpart. Having an implementation of Stacked Borrows also proved extremely valuable during development; it allowed us to quickly iterate with new rules and validate them against a few key test cases, easily discarding ideas that did not have the intended effect.

Implementation. Miri is an interpreter that operates on the MIR, an intermediate representation in the Rust compiler. MIR is an imperative, non-SSA, control-flow-graph based IR with a fairly small set of operations, which makes it well suited for an interpreter.

It was relatively straightforward to translate our operational rules into Rust code that runs whenever the interpreted program reads from or writes to memory. More interesting was the handling of **`retag`**; for this we decided to add a new primitive MIR statement and implemented a compiler pass (conceptually part of MIR construction in the compiler) that inserts **`retag`** statements automatically at the appropriate locations.

The implementation is fairly naive; the only optimization worth mentioning is that instead of storing a borrow stack per location, we store one stack for an adjacent range of locations that all share the same stack. Any memory access covering that entire range (say, a 4-byte access where these 4 locations share their stack) just performs the Stacked Borrows read/write action once. The ranges get automatically split and merged as the stack of adjacent locations diverges and re-unifies. As a consequence, memory storing, *e.g.*, an **`i32`** that is never subject to byte-wise accesses just needs a single borrow stack instead of 4 of them.

These changes have all been accepted into Miri, so running Miri now by default checks the program for conformity with Stacked Borrows. (Miri also checks for many other cases of undefined behavior, such as illegal use of uninitialized memory.) In the supplementary material [Jung et al. 2019], we provide links to an online version of Miri where the reader can run the optimization counterexamples from this paper and see Miri detecting the Stacked Borrows violations.

Testing and results. Miri is not very efficient (around 1000x slowdown compared to compiling the code with optimizations and running that), but it is fast enough to run some test suites. Concretely, we ran part of the Rust standard library test suites in Miri. We did not run the parts that are mostly concerned with host OS interaction (such as network and file system access) because Miri does not support those operations. The part that we *did* test includes key data structures (*e.g.*, `Vec`, `VecDeque`, `BTreeMap`, `HashMap`) as well as string formatting, arrays/slices, and iterator combinators. All of these involve interesting **`unsafe`** code using plenty of raw pointers.

Across all these tests, we found in total 7 cases of **`unsafe`** code not following Stacked Borrows. In two of these cases [Jung 2018a, 2019c], the code accidentally turned a shared reference into a mutable one. This is a pattern that the Rust developers explicitly forbade since day one; there is no question that such code is illegal, and thus patches to fix it were promptly accepted.

In three cases [Jung 2018b, 2019a,b], code created mutable references that were never used to access memory, but for Stacked Borrows, the mere fact that they exist already makes them in conflict with existing pointers (because creating a mutable reference is considered a write access). These could all be fixed by adapting the code, but one of them required some refactoring to use fewer mutable references and more raw pointers in `BTreeMap`. The fourth case [Jung 2019e] is similar, but this time was only visible to **`unsafe`** clients of `Vec`. These clients were making assumptions about `Vec` that are explicitly supported by the documentation, but were again violated by `Vec` internally creating (but not accessing) a conflicting mutable reference. We submitted patches to fix all of these cases, and they were all accepted. Still, this indicates that an interesting next step for Stacked Borrows is to be less aggressive about asserting uniqueness of mutable references that are created but never used.

The final case involves protectors [Jung 2019d]. That code passes around references wrapped inside a `struct`, and one function ends up invalidating these references while it is still ongoing, leading to an item with an active protector being popped. When we encountered this issue, instead of changing the code, we decided to adjust our model to accommodate this: we restricted `retag` to only operate on "bare" references, and not perform retagging of references inside compound types such as `struct`s. At that point it was unclear if the code should be considered correct or not, so this choice was helpful for us in order to focus our testing on other aspects of Stacked Borrows. Since then, however, after completing implementation and formalization of Stacked Borrows, new evidence has been discovered by the Rust community showing that this code is violating aliasing assumptions made by the compiler right now. What remains unclear is whether this should be fixed by changing the code, changing the alias analysis, or both (and giving the programmer more control). Miri will be a valuable tool to explore this trade-off in the future.

Miri is also available as a "Tool" on the Rust Playground, where small snippets of code can be tested directly in the browser.[12] Rust developers increasingly use Miri to check their code for undefined behavior, and if they are surprised by the result—*e.g.*, because code they deemed legal violates Stacked Borrows—we will often hear about that through bug reports or one of the various Rust community channels. So far, this has not uncovered any unsafe code patterns that are incompatible with Stacked Borrows beyond the ones discussed above.

We are now running Miri on the aforementioned Rust test suites every night, to continuously monitor the standard library for new cases of undefined behavior and Stacked Borrows violations. Some projects, including `HashMap` (which is a separate library that is also shipped as part of the Rust standard library), have also decided to run Miri as part of their continuous integration for pre-merge testing of pull requests.

Overall, we believe that this evaluation demonstrates that Stacked Borrows is a good fit for Rust. It might seem surprising that Rust developers would follow the stack discipline mandated by Stacked Borrows when mixing raw pointers and mutable references. Our hypothesis is that this works well because (a) raw pointers are currently untagged, so as long as *any* raw pointers may be used, *all* of them may be used, and (b) developers are aware that violating uniqueness of mutable references is not allowed, and already try as best they can to avoid it. Nevertheless, they currently do not know exactly what they can and cannot do. The goal of our work is to be able to give them a clearer answer to the questions that frequently arise in this space.

7.2 Coq Formalization

In this paper, we have given informal proof sketches of several representative optimizations enabled by Stacked Borrows. To further increase confidence in the semantics, we formalized its operational rules (6k lines of Coq, including proofs showing some key properties of the operational semantics) and turned our proof sketches into mechanized correctness proofs of all example transformations mentioned in this paper. To reason about transformations in the presence of unknown code, we built a simulation framework (5k lines of Coq) based on open simulations in the style of Hur et al. [2012]. See the supplementary material [Jung et al. 2019] for further details.

8 RELATED WORK

In terms of language semantics to enable better alias analysis, the most closely related to Stacked Borrows are C's *strict aliasing rules* and its `restrict` qualifier.

Strict aliasing rules. These rules, broadly speaking, allow the compiler to assume that pointers to different types do not alias. This is also often referred to as *type-based alias analysis* (TBAA). The C

[12] Available online at https://play.rust-lang.org/.

standard [ISO 2017] describes the strict aliasing rules in an axiomatic style. However, in particular the interaction of the strict aliasing rules with unions are not very clear in the standard. Under some conditions, the C standard permits "type-punning" through unions, meaning that a read with the "wrong" type is sometimes allowed, but the details are fuzzy.[13]

The first CompCert memory model [Leroy and Blazy 2008] formalizes a very strong operational version of the strict aliasing rules that entirely disallows type-punning through unions. However, this is not exploited for the purpose of program analyses or transformations. (Later versions of CompCert use a simpler memory model that does not impose any strict aliasing rules [Leroy et al. 2012].) Krebbers [2013] gives another operational account of strict aliasing, with rules for type-punning through unions that are based on the GCC developers' interpretation of the C standard. He also shows a basic non-aliasing "soundness" theorem, but no compiler transformations.

Long-standing compiler bugs in both GCC [Hazeghi 2013] and clang [Fraine 2014] indicate that exploiting strict aliasing rules for optimizations is tricky and easily leads to miscompilations. Moreover, many C programmers consider the strict aliasing rules to be overly restrictive, leading to many strict-aliasing violations in real-world C code [Memarian and Sewell 2016]. For these reasons, many large projects, including the Linux kernel, outright disable type-based alias analysis, essentially opting-in to a version of C with less undefined behavior and fewer optimizations.

Moreover, type-based alias analysis is comparatively weak. In particular, it cannot be used to reason about unknown code; the compiler must know the types of both memory accesses involved to determine if they might alias. In contrast, as we have shown, Stacked Borrows enables optimizations involving unknown code.

The only tool for detecting at least some violations of strict aliasing rules that we are aware of is libcrunch[14], which however is neither sound nor complete [Regehr 2017].

`restrict`-*qualified pointers.* Since C99, the C language knows the `restrict` qualifier for pointers, an explicit annotation that can be used by the programmer to give non-aliasing information to the compiler. This qualifier indicates that accesses performed through this pointer and pointers derived from it cannot alias with other accesses. One common application of `restrict` is in tight numerical loops, *e.g.*, matrix multiplication, where assuming that the output matrix does not alias either of the input matrices can make the difference between a fully vectorized loop using SIMD (single instruction multiple data) operations and purely scalar (unvectorized) code.

Conceptually, `restrict` is closely related to Stacked Borrows. In fact, the Rust compiler (which uses LLVM as its backend) used to emit the LLVM equivalent of `restrict` as annotations for mutable references in function argument position.

However, the exact semantics of `restrict` is unclear, in particular when considering general pointers and not just function arguments. (Function arguments are easier because there is a clear notion of "scope" that one could use to say for *how long* the aliasing guarantee must hold.) Even for function arguments, uncertainty in the semantics led to several LLVM bugs [Gohman 2015; Popov 2018], due to which this annotation is currently *not* emitted by Rust for mutable references.

We are not aware of any operational formalization of `restrict`, nor any tool to check for violations of its properties, but we could imagine that some of the ideas from Stacked Borrows could be useful in such a project.

Fortran. Loosely related to C's `restrict` are the aliasing rules of Fortran [ANSI 1978], which disallow function parameters to alias (unless the programmer specifically marks them as potentially

[13] The standard says that such type-punning reads are permitted through a union if several conditions all apply, including "that a declaration of the completed type of the union is visible". It is not clear exactly what that means.
[14] Available online at https://github.com/stephenrkell/libcrunch.

aliasing). In fact, competing with Fortran compilers was a primary motivation for adding `restrict` to C [Drepper 2007]. Nguyen and Irigoin [2003] describe a tool that dynamically checks for aliasing violations in Fortran programs, but they do not verify any program transformations.

Low-level language semantics. There is a large body of work on formalizing the semantics of C or LLVM (as representative examples of highly optimized "low-level" languages) and in particular their handling of pointers and pointer provenance [Memarian et al. 2019; Krebbers 2015; Kang et al. 2015; Lee et al. 2018; Hathhorn et al. 2015; Norrish 1998]. However, with the exception of what was explained above, this work does not account for strict aliasing rules and the `restrict` qualifier. They, instead, focus on orthogonal aspects, such as the handling of casts between integers and pointers, and the use of pointer provenance to prevent pointer arithmetic across object boundaries.

We have specifically designed Stacked Borrows to *not* assume that all pointers have a known provenance, by adding the notion of an "untagged" pointer. This means we should be able to basically take any of the existing approaches to model integer-pointer casts, and equip it with a variant of Stacked Borrows that handles pointers of unknown provenance as untagged.

The CompCert compiler performs an alias analysis that has been formally verified [Robert and Leroy 2012]. However, that analysis does not exploit extra aliasing information provided by the language.

9 CONCLUSION

We have described Stacked Borrows, an operational semantics encoding aliasing rules for Rust's reference types. Our goal was to exploit the rich alias information encoded in those types in order to enable *intraprocedural* optimizations, whose validity we have proved formally in Coq.

The core idea of the model is to implement a dynamic version of the borrow checker, the part of the Rust type checker that enforces the aliasing rules for references in safe code. This allows us to extend the scope of the analysis to also cover **`unsafe`** code manipulating raw pointers, essentially "extrapolating" borrow checking from the safe fragment of Rust to the full language. We have implemented this model in an interpreter, Miri, which we have run on large parts of the standard library test suite to verify that, indeed, it adequately captures how **`unsafe`** Rust code gets written.

In terms of future work, our evaluation with Miri identified two regular patterns of Stacked Borrows violations (conflicting mutable references being created but not used, and references in private fields being guarded by protectors). We plan to study if it is possible to accommodate these patterns without giving up undue amounts of optimization potential. We suspect a more tree-based structure as mentioned in §3.3 could help with the issue for mutable references in particular. It would also be interesting to run Miri on a larger body of Rust code.

In addition, we plan to connect Stacked Borrows with the formal Rust type system of RustBelt [Jung et al. 2018] to verify that in fact all safe Rust programs comply with Stacked Borrows.

Finally, we intend to continue to work closely with the Rust community and development teams to adjust Stacked Borrows to their needs, with the goal of eventually making a variant of Stacked Borrows part of the official semantics of Rust.

ACKNOWLEDGMENTS

We wish to thank the Rust community and in particular Nicholas Matsakis for their feedback and discussions that helped make Stacked Borrows align with Rust developers' intuition.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

REFERENCES

ANSI. 1978. *Programming Language FORTRAN.* ANSI X3.9-1978.

Ulrich Drepper. 2007. Memory part 5: What programmers can do. LWN article. https://lwn.net/Articles/255364/.

Bruno De Fraine. 2014. Wrong results with union and strict-aliasing. LLVM issue #21725. https://bugs.llvm.org/show_bug.cgi?id=21725.

Rakesh Ghiya, Daniel M. Lavery, and David C. Sehr. 2001. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *PLDI*. 47–58. https://doi.org/10.1145/378795.378806

Dan Gohman. 2015. Incorrect liveness in DeadStoreElimination. LLVM issue #25422. https://bugs.llvm.org/show_bug.cgi?id=25422.

Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 336–345. https://doi.org/10.1145/2737924.2737979

Dara Hazeghi. 2013. Store motion causes wrong code for union access at -O3. GCC issue #57359. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=57359.

Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *TOPLAS* 19, 1 (1997), 1–6. https://doi.org/10.1145/239912.239913

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *POPL*. https://doi.org/10.1145/2103656.2103666

ISO. 2017. *C17 Standard.* ISO/IEC 9899:2018.

Ralf Jung. 2018a. Fix futures creating aliasing mutable and shared ref. Rust pull request #56319. https://github.com/rust-lang/rust/pull/56319.

Ralf Jung. 2018b. VecDeque: fix for stacked borrows. Rust pull request #56161. https://github.com/rust-lang/rust/pull/56161.

Ralf Jung. 2019a. Fix LinkedList invalidating mutable references. Rust pull request #60072. https://github.com/rust-lang/rust/pull/60072.

Ralf Jung. 2019b. Fix overlapping references in BTree. Rust pull request #58431. https://github.com/rust-lang/rust/pull/58431.

Ralf Jung. 2019c. Fix str mutating through a ptr derived from &self. Rust pull request #58200. https://github.com/rust-lang/rust/pull/58200.

Ralf Jung. 2019d. VecDeque's Drain::drop writes to memory that a shared reference points to. Rust issue #60076. https://github.com/rust-lang/rust/issues/60076.

Ralf Jung. 2019e. Vec::push invalidates interior references even when it does not reallocate. Rust issue #60847. https://github.com/rust-lang/rust/issues/60847.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: An aliasing model for Rust – Artifact. https://doi.org/10.5281/zenodo.3541779 (latest version available on paper website: https://plv.mpi-sws.org/rustbelt/stacked-borrows/).

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL, Article 66 (2018).

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *PLDI*. 326–335. https://doi.org/10.1145/2737924.2738005

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* https://doc.rust-lang.org/stable/book/2018-edition/

Felix S. Klock. 2019. Breaking news: Non-lexical lifetimes arrives for everyone. Blog post. http://blog.pnkfx.org/blog/2019/06/26/breaking-news-non-lexical-lifetimes-arrives-for-everyone/.

Robbert Krebbers. 2013. Aliasing restrictions of C11 formalized in Coq. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. 50–65. https://doi.org/10.1007/978-3-319-03545-1_4

Robbert Krebbers. 2015. *The C standard formalized in Coq.* Ph.D. Dissertation. Radboud University.

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *PACMPL* 2, OOPSLA (2018), 125:1–125:28. https://doi.org/10.1145/3276495

Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert memory model, version 2.* Technical Report RR-7987. Inria.

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Niko Matsakis. 2018. An alias-based formulation of the borrow checker. Blog post. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *PACMPL* 3, POPL (2019), 67:1–67:32. https://doi.org/10.1145/3290380

Kayvan Memarian and Peter Sewell. 2016. N2014: What is C in practice? (Cerberus survey v2): Analysis of Responses. ISO SC22 WG14 N2014, http://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html.

Thi Viet Nga Nguyen and François Irigoin. 2003. Alias verification for Fortran code optimization. *J. UCS* 9, 3 (2003), 270. https://doi.org/10.3217/jucs-009-03-0270

Michael Norrish. 1998. *C formalised in HOL*. Ph.D. Dissertation. University of Cambridge.

Nikita Popov. 2018. Loop unrolling incorrectly duplicates noalias metadata. LLVM issue #39282. https://bugs.llvm.org/show_bug.cgi?id=39282.

John Regehr. 2017. Undefined behavior in 2017. Blog post. https://blog.regehr.org/archives/1520.

Valentin Robert and Xavier Leroy. 2012. A formally-verified alias analysis. In *CPP*. 11–26. https://doi.org/10.1007/978-3-642-35308-6_5

Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *PLDI*. 1. https://doi.org/10.1145/207110.207111