Rust's Unsafe Code Guidelines Reference

This document is a past effort by the UCG WG to provide a "guide" for writing unsafe code that "recommends" what kinds of things unsafe code can and cannot do, and that documents which guarantees unsafe code may rely on. It is largely abandoned right now. However, the glossary is actively maintained.

Unless stated otherwise, the information in the guide is mostly a "recommendation" and still subject to change.

Glossary

ABI (of a type)

The *function call ABI* or short *ABI* of a type defines how it is passed *by-value* across a function boundary. Possible ABIs include passing the value directly in zero or more registers, or passing it indirectly as a pointer to the actual data. The space of all possible ABIs is huge and extremely target-dependent. Rust therefore does generally not clearly define the ABI of any type, it only defines when two types are *ABI-compatible*, which means that it is legal to call a function declared with an argument or return type `T` using a declaration or function pointer with argument or return type `U`.

Note that ABI compatibility is stricter than layout compatibility. For instance `#[repr(C)] struct S(i32)` is (guaranteed to be) layout-compatible with `i32`, but it is *not* ABI-compatible.

Abstract Byte

The *byte* is the smallest unit of storage in Rust. Memory allocations are thought of as storing a list of bytes, and at the lowest level each load returns a list of bytes and each store takes a list of bytes and puts it into memory. (The representation relation then defines how to convert between those lists of bytes and higher-level values such as mathematical integers or pointers.)

However, a *byte* in the Rust Abstract Machine is more complicated than just an integer in `0..256` -- think of it as there being some extra "shadow state" that is relevant for the Abstract Machine execution (in particular, for whether this execution has UB), but that disappears when compiling the program to assembly. That's why we call it *abstract byte*, to distinguish it from the physical machine byte in `0..256`.

The most obvious "shadow state" is tracking whether memory is initialized. See this blog post for details, but the gist of it is that bytes in memory are more like `Option<u8>` where `None` indicates that this byte is uninitialized. Operations like `copy` work on that representation, so if you copy from some uninitialized memory into initialized memory, the target memory becomes "de-initialized". Another piece of shadow state is pointer

provenance: the Abstract Machine tracks the "origin" of each pointer value to enforce the rule that a pointer used to access some memory is "based on" the original pointer produced when that memory got allocated. This provenance must be preserved when the pointer is stored to memory and loaded again later, which implies that abstract bytes must be able to carry provenance.

Without committing to the exact shape of provenance in Rust, we can therefore say that an (abstract) byte in the Rust Abstract Machine looks as follows:

```
pub enum AbstractByte<Provenance> {
    /// An uninitialized byte.
    Uninit,
    /// An initialized byte with a value in `0..256`,
     /// optionally with some provenance (if it is encoding a
pointer).
    Init(u8, Option<Provenance>),
}
```

Aliasing

Aliasing occurs when one pointer or reference points to a "span" of memory that overlaps with the span of another pointer or reference. A span of memory is similar to how a slice works: there's a base byte address as well as a length in bytes.

Note: a full aliasing model for Rust, defining when aliasing is allowed and when not, has not yet been defined. The purpose of this definition is to define when aliasing *happens*, not when it is *allowed*. The most developed potential aliasing model so far is Stacked Borrows.

Consider the following example:

```
fn main() {
    let u: u64 = 7_u64;
    let r: &u64 = &u;
    let s: &[u8] = unsafe {
        core::slice::from_raw_parts(&u as *const u64 as *const
u8, 8)
```

```
    };
    let (head, tail) = s.split_first().unwrap();
}
```

In this case, both `r` and `s` alias each other, since they both point to all of the bytes of `u`.

However, `head` and `tail` do not alias each other: `head` points to the first byte of `u` and `tail` points to the other seven bytes of `u` after it. Both `head` and `tail` alias `s`, any overlap is sufficient to count as an alias.

The span of a pointer or reference is the size of the value being pointed to or referenced. Depending on the type, you can determine the size as follows:

- For a type `T` that is `Sized` The span length of a pointer or reference to `T` is found with `size_of::<T>()`.
- When `T` is not `Sized` the story is a little tricker:
    - If you have a reference `r` you can use `size_of_val(r)` to determine the span of the reference.
    - If you have a pointer `p` you must unsafely convert that to a reference before you can use `size_of_val`. There is not currently a safe way to determine the span of a pointer to an unsized type.

The Data layout chapter also has more information on the sizes of different types.

One interesting side effect of these rules is that references and pointers to Zero Sized Types *never* alias each other, because their span length is always 0 bytes.

It is also important to know that LLVM IR has a `noalias` attribute that works somewhat differently from this definition. However, that's considered a low level detail of a particular Rust implementation. When programming Rust, the Abstract Rust Machine is intended to operate according to the definition here.

Allocation

An *allocation* is a chunk of memory that is addressable from Rust. Allocations are created for objects on the heap, for stack-allocated variables, for globals (statics and consts), but also for objects that do not have Rust-inspectable data such as functions and vtables. An allocation has a contiguous range of memory addresses that it covers, and it can generally only be deallocated all at once. (Though in the future, we might allow allocations with holes, and we might allow growing/shrinking an allocation.) This range can be empty, but even empty allocations have a *base address* that they are located at. The base address of an allocation is not necessarily unique; but if two distinct allocations have the same base address then at least one of them must be empty.

Pointer arithmetic is generally only possible within an allocation: provenance ensures that each pointer "remembers" which allocation it points to, and accesses are only permitted if the address is in range of the allocation associated with the pointer.

Data inside an allocation is stored as abstract bytes; in particular, allocations do not track which type the data inside them has.

Interior mutability

Interior Mutation means mutating memory where there also exists a live shared reference pointing to the same memory; or mutating memory through a pointer derived from a shared reference. "live" here means a value that will be "used again" later. "derived from" means that the pointer was obtained by casting a shared reference and potentially adding an offset. This is not yet precisely defined, which will be fixed as part of developing a precise aliasing model.

Finding live shared references propagates recursively through references, but not through raw pointers. So, for example, if data immediately pointed to by a `&T` or `& &mut T` is mutated, that's interior mutability. If data immediately pointed to by a `*const T` or `&*const T` is mutated, that's *not* interior mutability.

Interior mutability refers to the ability to perform interior mutation without causing UB. All interior mutation in Rust has to happen inside an

`UnsafeCell`, so all data structures that have interior mutability must (directly or indirectly) use `UnsafeCell` for this purpose.

Layout

The *layout* of a type defines its size and alignment as well as the offsets of its subobjects (e.g. fields of structs/unions/enums/... or elements of arrays, and the discriminant of enums).

Note that layout does not capture everything that there is to say about how a type is represented on the machine; it notably does not include ABI or Niches.

Note: Originally, *layout* and *representation* were treated as synonyms, and Rust language features like the `#[repr]` attribute reflect this. In this document, *layout* and *representation* are not synonyms.

Memory Address

A *memory address* is an integer value that identifies where in the process' memory some data is stored. This will typically be a virtual address, if the Rust process runs as a regular user-space program. It can also be a physical address for bare-level / kernel code. Rust doesn't really care either way, the point is: it's an address as understood by the CPU, it's what the load/store instructions need to identify where in memory to perform the load/store.

Note that a pointer in Rust is *not* just a memory address. A pointer value consists of a memory address and provenance.

Niche

The *niche* of a type determines invalid bit-patterns that will be used by layout optimizations.

For example, `&mut T` has at least one niche, the "all zeros" bit-pattern. This niche is used by layout optimizations like "`enum` discriminant elision" to guarantee that `Option<&mut T>` has the same size as `&mut T`.

While all niches are invalid bit-patterns, not all invalid bit-patterns are niches. For example, the "all bits uninitialized" is an invalid bit-pattern for

`&mut T`, but this bit-pattern cannot be used by layout optimizations, and is not a niche.

Padding

Padding (of a type `T`) refers to the space that the compiler leaves between fields of a struct or enum variant to satisfy alignment requirements, and before/after variants of a union or enum to make all variants equally sized.

Padding can be thought of as the type containing secret fields of type `[Pad; N]` for some hypothetical type `Pad` (of size 1) with the following properties:

- `Pad` is valid for any byte, i.e., it has the same validity invariant as `MaybeUninit<u8>`.
- Copying `Pad` ignores the source byte, and writes *any* value to the target byte. Or, equivalently (in terms of Abstract Machine behavior), copying `Pad` marks the target byte as uninitialized.

Note that padding is a property of the *type* and not the memory: reading from the padding of an `&Foo` (by casting to a byte reference) may produce initialized values if the `&Foo` is pointing to memory that was initialized (for example, if it was originally a byte buffer initialized to `0`), but the moment you perform a typed copy out of that reference you will have uninitialized padding bytes in the copy.

We can also define padding in terms of the representation relation: A byte at index `i` is a padding byte for type `T` if, for all values `v` and lists of bytes `b` such that `v` and `b` are related at `T` (let's write this `Vrel_T(v, b)`), changing `b` at index `i` to any other byte yields a `b'` such `v` and `b'` are related (`Vrel_T(v, b')`). In other words, the byte at index `i` is entirely ignored by `Vrel_T` (the value relation for `T`), and two lists of bytes that only differ in padding bytes relate to the same value(s), if any.

This definition works fine for product types (structs, tuples, arrays, ...). The desired notion of "padding byte" for enums and unions is still unclear.

Place

A *place* (called "lvalue" in C and "glvalue" in C++) is the result of computing a *place expression*. A place is basically a pointer (pointing to some location in memory, potentially carrying provenance), but might contain more information such as size or alignment (the details will have to be determined as the Rust Abstract Machine gets specified more precisely). A place has a type, indicating the type of values that it stores.

The key operations on a place are:

- Storing a value of the same type in it (when it is used on the left-hand side of an assignment).
- Loading a value of the same type from it (through the place-to-value coercion).
- Converting between a place (of type `T`) and a pointer value (of type `&T`, `&mut T`, `*const T` or `*mut T`) using the `&` and `*` operators. This is also the only way a place can be "stored": by converting it to a value first.

Pointer Provenance

The *provenance* of a pointer is used to distinguish pointers that point to the same memory address (i.e., pointers that, when cast to `usize`, will compare equal). Provenance is extra state that only exists in the Rust Abstract Machine; it is needed to specify program behavior but not present any more when the program runs on real hardware. In other words, pointers that only differ in their provenance can *not* be distinguished any more in the final binary (but provenance can influence how the compiler translates the program).

The exact form of provenance in Rust is unclear. It is also unclear whether provenance applies to more than just pointers, i.e., one could imagine integers having provenance as well (so that pointer provenance can be preserved when pointers are cast to an integer and back). In the following, we give some examples if what provenance *could* look like.

Using provenance to track originating allocation. For example, we have to distinguish pointers to the same location if they originated from different

allocations. Cross-allocation pointer arithmetic does not lead to usable pointers, so the Rust Abstract Machine *somehow* has to remember the original allocation to which a pointer pointed. It could use provenance to achieve this:

```
// Let's assume the two allocations here have base addresses
0x100 and 0x200.
// We write pointer provenance as `@N` where `N` is some kind
of ID uniquely
// identifying the allocation.
let raw1 = Box::into_raw(Box::new(13u8));
let raw2 = Box::into_raw(Box::new(42u8));
let raw2_wrong = raw1.wrapping_add(raw2.wrapping_sub(raw1 as
usize) as usize);
// These pointers now have the following values:
// raw1 points to address 0x100 and has provenance @1.
// raw2 points to address 0x200 and has provenance @2.
// raw2_wrong points to address 0x200 and has provenance @1.
// In other words, raw2 and raw2_wrong have same *address*...
assert_eq!(raw2 as usize, raw2_wrong as usize);
// ...but it would be UB to dereference raw2_wrong, as it has
the wrong *provenance*:
// it points to address 0x200, which is in allocation @2, but
the pointer
// has provenance @1.
```

This kind of provenance also exists in C/C++, but Rust is more permissive by (a) providing a way to do pointer arithmetic across allocation boundaries without causing immediate UB (though, as we have seen, the resulting pointer still cannot be used for locations outside the allocation it originates), and (b) by allowing pointers to always be compared safely, even if their provenance differs. For some more information, see this document proposing a more precise definition of provenance for C.

Using provenance for Rust's aliasing rules. Another example of pointer provenance is the "tag" from Stacked Borrows. For some more information, see this blog post.

Representation (relation)

A *representation* of a value is a list of (abstract) bytes that is used to store or "represent" that value in memory.

We also sometimes speak of the *representation of a type*; this should more correctly be called the *representation relation* as it relates values of this type to lists of bytes that represent this value. The term "relation" here is used in the mathematical sense: the representation relation is a predicate that, given a value and a list of bytes, says whether this value is represented by that list of bytes (`val -> list byte -> Prop`).

The relation should be functional for a fixed list of bytes (i.e., every list of bytes has at most one associated representation). It is partial in both directions: not all values have a representation (e.g. the mathematical integer `300` has no representation at type `u8`), and not all lists of bytes correspond to a value of a specific type (e.g. lists of the wrong size correspond to no value, and the list consisting of the single byte `0x10` corresponds to no value of type `bool`). For a fixed value, there can be many representations (e.g., when considering type `#[repr(C)] Pair(u8, u16)`, the second byte is a padding byte so changing it does not affect the value represented by a list of bytes).

See the value domain for an example how values and representation relations can be made more precise.

Soundness (of code / of a library)

Soundness is a type system concept (actually originating from the study of logics) and means that the type system is "correct" in the sense that well-typed programs actually have the desired properties. For Rust, this means well-typed programs cannot cause Undefined Behavior. This promise only extends to safe code however; for `unsafe` code, it is up to the programmer to uphold this contract.

Accordingly, we say that a library (or an individual function) is *sound* if it is impossible for safe code to cause Undefined Behavior using its public API. Conversely, the library/function is *unsound* if safe code *can* cause Undefined Behavior.

Undefined Behavior

Undefined Behavior is a concept of the contract between the Rust programmer and the compiler: The programmer promises that the code exhibits no undefined behavior. In return, the compiler promises to compile the code in a way that the final program does on the real hardware what the source program does according to the Rust Abstract Machine. If it turns out the program *does* have undefined behavior, the contract is void, and the program produced by the compiler is essentially garbage (in particular, it is not bound by any specification; the program does not even have to be well-formed executable code).

In Rust, the Nomicon and the Reference both have a list of behavior that the language considers undefined. Rust promises that safe code cannot cause Undefined Behavior---the compiler and authors of unsafe code takes the burden of this contract on themselves. For unsafe code, however, the burden is still on the programmer.

Also see: Soundness.

Validity and safety invariant

The *validity invariant* is an invariant that all data must uphold any time it is accessed or copied in a typed manner. This invariant is known to the compiler and exploited by optimizations such as improved enum layout or eliding in-bounds checks.

In terms of MIR statements, "accessed or copied" means whenever an assignment statement is executed. That statement has a type (LHS and RHS must have the same type), and the data being assigned must be valid at that type. Moreover, arguments passed to a function must be valid at the type given in the callee signature, and the return value of a function must be valid at the type given in the caller signature. OPEN QUESTION: Are there more cases where data must be valid?

In terms of code, some data computed by `TERM` is valid at type `T` if and only if the following program does not have UB:

```
fn main() { unsafe {
  let t: T = std::mem::transmute(TERM);
} }
```

The *safety* invariant is an invariant that safe code may assume all data to uphold. This invariant is used to justify which operations safe code can perform. The safety invariant can be temporarily violated by unsafe code, but must always be upheld when interfacing with unknown safe code. It is not relevant when arguing whether some *program* has UB, but it is relevant when arguing whether some code safely encapsulates its unsafety -- in other words, it is relevant when arguing whether some *library* is sound.

In terms of code, some data computed by `TERM` (possibly constructed from some `arguments` that can be *assumed* to satisfy the safety invariant) is valid at type `T` if and only if the following library function can be safely exposed to arbitrary (safe) code as part of the public library interface:

```
pub fn make_something(arguments: U) -> T { unsafe {
  std::mem::transmute(TERM)
} }
```

One example of valid-but-unsafe data is a `&str` or `String` that's not well-formed UTF-8: the compiler will not run its own optimizations that would cause any trouble here, so unsafe code may temporarily violate the invariant that strings are `UTF-8`. However, functions on `&str`/`String` may assume the string to be `UTF-8`, meaning they may cause UB if the string is *not* `UTF-8`. This means that unsafe code violating the UTF-8 invariant must not perform string operations (it may operate on the data as a byte slice though), or else it risks UB. Moreover, such unsafe code must not return a non-UTF-8 string to the "outside" of its safe abstraction boundary, because that would mean safe code could cause UB by doing `bad_function().chars().count()`.

To summarize: *Data must always be valid, but it only must be safe in safe code.* For some more information, see this blog post.

Value

A *value* (called "value of the expression" or "rvalue" in C and "prvalue" in C++) is what gets stored in a place, and also the result of computing a *value expression*. A value has a type, and it denotes the abstract mathematical concept that is represented by data in our programs.

For example, a value of type `u8` is a mathematical integer in the range `0..256`. Values can be (according to their type) turned into a list of (abstract) bytes, which is called a representation of the value. Values are ephemeral; they arise during the computation of an instruction but are only ever persisted in memory through their representation. (This is comparable to how run-time data in a program is ephemeral and is only ever persisted in serialized form.)

Zero-sized type / ZST

Types with zero size are called zero-sized types, which is abbreviated as "ZST". This document also uses the "1-ZST" abbreviation, which stands for "one-aligned zero-sized type", to refer to zero-sized types with an alignment requirement of 1.

For example, `()` is a "1-ZST" but `[u16; 0]` is not because it has an alignment requirement of 2.

Data layout

Layout of structs and tuples

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Layout of scalar types

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Layout of Rust `enum` types

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Layout of unions

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Layout of reference and pointer types

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Representation of Function Pointers

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Layout of Rust array types and slices

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Layout of packed SIMD vectors

This page has been archived

It did not actually reflect current layout guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Validity

Validity of unions

This page has been archived

It did not actually reflect current language guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Validity of function pointers

This page has been archived

It did not actually reflect current language guarantees and caused frequent confusion.

The old content can be accessed on GitHub.

Optimizations

This page has been archived

It did not actually reflect current language guarantees and caused frequent confusion.

The old content can be accessed on GitHub.