1. Undefined Behavior as a Domain Restriction

1.1. TL;DR

Undefined behavior is not "wrong output."
It means the program has stepped *outside the compiler's contract.*
Unreachable UB is allowed; reachable UB voids all guarantees.

This document tries to capture the Rektoff Cohort 3, office hours conversation around Rust's undefined behavior.

1.2. Model

Let:

- P := the set of all **syntactically valid** Rust programs
- $S \subset P$:= programs for which **no possible execution** can trigger UB
- $U := P \setminus S$
- C := the set of machine-code executables

Model the compiler as a **partial function**:

$\text{compile} : S \to C$

It is defined only on S. Programs in U lie outside the compiler's domain of definition.

The compiler maps valid Rust programs to well-defined executables.

1.3. Reachability Is the Boundary

Membership in S is a property of **executions** (e), not syntax.

Formally:

$$p \in S \quad \text{if} \quad \forall e \in \text{Exec}(p), e \text{ does not trigger UB}$$

Unreachable undefined behavior is permitted; reachable undefined behavior is not.

1.4. Why This Matters

1.4.1. Unreachable UB: allowed (in S)

```
if false {
    unsafe {
        *(0 as *mut u8) = 1;
    }
}
```

- Syntactically valid (in P)
- The branch is provably unreachable
- No execution can trigger UB
- Therefore the program is in S, and compile is well-defined

1.4.2. Reachable UB: forbidden (in U)

```
if cond { // get rekt!
    unsafe {
        *(0 as *mut u8) = 1;
    }
}
```

- There exists an execution where `cond == true`
- UB is reachable
- Therefore the program is in U

- The compiler’s semantic guarantees no longer apply

This holds even if cond is “always false in practice”.

1.5. Diagram (Mental Model)

```
P  (syntactically valid programs)
┌────────────────────────────────────┐
│                                    │
│   S  (all executions UB-free)      │
│   ┌────────────────────────────┐   │
│   │                            │   │
│   │   compile : S -> C         │   │
│   │                            │   │
│   └────────────────────────────┘   │
│                                    │
│   U = P - S  (reachable UB)        │
│   - outside compiler contract      │
│   - no semantic guarantees         │
│                                    │
└────────────────────────────────────┘
```

Only programs inside S are valid inputs to the compiler-as-a-function.

1.6. Consequences

UB $\neq$ incorrect output

UB = execution outside the compiler's contract

If a program admits an execution that triggers UB:

- The program lies outside the compiler’s domain of definition
- The compiler always assumes its input satisfied Rust’s rules
- No semantic guarantees apply to the resulting behavior

1.7. Contrast: Rust vs C/C++

Key difference: where the boundary is enforced

Rust

- UB is defined in terms of possible executions
- Unreachable UB is explicitly allowed
- Safety rules are designed to preserve soundness under aggressive optimization (the whole pipeline of AST -> MIR -> … LLVM) operates on trusted inputs. GIGO caveats.

C / C++

- UB is often tied more loosely to syntax and local reasoning
- “This code never runs” arguments are more fragile. (Oh yea? prove it!)
- Compilers may generate UB in ways that surprise even experienced developers

Put differently:

> Rust draws a hard semantic boundary around what may execute. C and C++ often leave that boundary implicit, and leaky.

1.8. Conclusion

> A Rust program is valid only if every possible execution stays within the compiler’s assumptions.

1.9. The obvious lingering question:

- How do you prove a rust program has no UB?