A Geometric Framework for Undefined Behavior and Lifetime Semantics in Systems Languages

Ehsan M. Kermani[*]

Abstract

Undefined Behavior (UB) is a central challenge in systems programming languages such as Rust, C++ and Mojo. It appears when a program enters a configuration that violates the language's memory model, for example use after free, iterator invalidation or illegal aliasing. Existing approaches to UB are mostly syntactic or logical (type systems, separation logic, abstract interpretation), and they tend to be tailored to a single language.

This paper proposes a geometric and categorical framework for *memory* related UB. We model abstract program states ("worlds") as a poset, equip this poset with the Alexandroff topology, and interpret safe executions as upward closed subsets. UB then shows up as *holes* in the safe state space: non fillable cycles or missing higher dimensional cells. These holes are detected by simplicial homology and relative homology in the sense of classical algebraic topology [1]. On top of this, we give a categorical semantics where lifetimes form a category, reference types are bifunctors, and reborrows correspond to natural transformations in the sense of Mac Lane [5].

We illustrate the framework on three languages: Rust, C++ and Mojo. For Rust we work out a concrete example where a familiar UB pattern produces a one dimensional homology class. For C++ we build a toy complex for a function that combines raw pointer invalidation and iterator invalidation, and we show that its first homology group is isomorphic to $\mathbb{Z}^2$, reflecting two independent UB cavities. For Mojo we sketch how origin indexed types and origin unions fit into the same picture, and how cohomological obstructions on a poset of origins can express global inconsistencies. Finally, we describe a high level algorithm for homology based UB detection and discuss possible applications to static analysis, linting and language design.

1 Introduction

Systems languages such as Rust, C++ and Mojo try to combine low-level control with strong safety guarantees. A central obstacle is *Undefined Behavior* (UB): program configurations in which the language semantics makes no promises. Examples include

- dereferencing pointers after their referents are freed (use-after-free),
- reusing invalidated iterators or views,
- violating aliasing rules, for instance concurrent mutable accesses,
- misusing low-level intrinsics or unsafe blocks.

In this paper we use the word *safety* in a specific and narrow sense. Unless stated otherwise, safety means *memory safety*: the absence of behaviors such as use after free, out of bounds access, invalidated views or iterators, and violations of aliasing and lifetime disciplines. Other important

[*]This research was conducted independently, outside the scope of my employment. For correspondence: ehsan@modular.com

properties, such as type safety, data race freedom or progress, are outside the scope of this initial work.

Rust avoids UB in *safe* code through its ownership and borrowing system. C++ leaves more responsibility to the programmer and to external tools. Mojo introduces origins and lifetimes that are inspired by Rust but use a different surface design [7, 8, 9]. Formal models such as RustBelt [4], Oxide [10], Cerberus [6] and Stacked Borrows [3] use logical and operational techniques to show that well-typed programs do not exhibit UB. These approaches are powerful, but they do not explicitly describe the global shape of the space of abstract states, nor do they present UB as a topological obstruction in that space.

This paper suggests a complementary point of view: **UB has a geometric shape**. One can think of the set of all abstract program states as a space built from points (states), edges (transitions) and higher-dimensional cells (commuting squares, cubes and so on). Safe executions occupy a "filled-in" region of this space. UB corresponds to missing cells and non-contractible loops, that is, to topological holes.

We make this picture precise as follows.

- Abstract states form a poset of *worlds* ordered by safe extension.
- This poset carries the Alexandroff topology, so upward-closed sets are open.
- From worlds and transitions we build a simplicial complex.
- Homology and relative homology reveal holes in this complex [1].
- Lifetimes and reference types are described categorically [5], with references as bifunctors and reborrows as natural transformations.

In the rest of the paper we apply this idea to lifetime and ownership semantics in Rust, C++ and Mojo. For Rust we show a small example where a familiar UB pattern produces a one-dimensional hole in the state space.

Contributions. This paper makes the following contributions.

- We propose a geometric view of UB in systems languages, based on a poset of abstract worlds, its Alexandroff topology, and simplicial complexes that capture safe transitions.
- We give a categorical account of lifetimes and references in terms of a lifetime category, a type category, and a bifunctor that models reference types, and we explain how reborrows correspond to natural transformations.
- We work out a concrete Rust example in which a familiar UB pattern produces a one dimensional homology class, and we sketch how analogous structures arise in C++ and Mojo.
- We present a high level algorithm for homology based detection of candidate UB patterns and discuss how it could be integrated with existing compiler and analysis pipelines.

We do not claim a complete or sound characterization of all forms of UB. The goal is to introduce a language independent geometric layer that can complement existing semantic and type theoretic approaches.

2 Problem Statement

We are interested in a framework that satisfies the following goals.

(1) **Language independent states.** We would like a common notion of abstract state that can speak about Rust's borrow stacks, C++ raw pointers and Mojo origins.

(2) **Structural explanation of UB.** Instead of treating UB as a collection of prohibited patterns, we want to explain why certain patterns are fundamentally unsafe, namely because they correspond to holes in the state space.

(3) **Computable invariants.** We want algebraic invariants, such as homology and relative homology, that can be approximated or computed on state graphs extracted from code, at least on bounded scopes [1].

(4) **Compatibility with type systems.** The framework should complement existing type-based techniques. For example, one can view Rust's borrow checker and models such as RustBelt [4] and Stacked Borrows [3] as mechanisms that enforce the absence of particular holes.

To the best of our knowledge, there is no current semantics of Rust, C++ or Mojo that uses topological invariants to reason about UB in this way. Our goal is to make that connection explicit and use it as a design tool.

3 Preliminaries

3.1 Posets and the Alexandroff Topology

Let $(W, \leq)$ be a partially ordered set of *worlds*, which represent abstract program states. Intuitively, $w \leq w'$ means that w' is a safe extension of w: we have executed some additional steps, or allocated new memory, without breaking any invariants.

The *Alexandroff topology* (also called the specialization topology) [1] on W is defined by

$$U \in \tau_W \iff \forall w \in U,\ \forall w' \in W,\ (w \leq w') \Rightarrow w' \in U.$$

In other words, the open sets are exactly the upward-closed subsets of W.

This topology is natural for program states because it reflects the idea that execution is monotone in the abstraction: once a state is safe, any state that extends it by performing more safe steps should also be considered safe. Upward closed sets express properties that are preserved by continuing safe execution, which is exactly the form of monotonicity that many abstract domains and dataflow analyses rely on.

We will use the following interpretation.

- W is the space of all abstract states that we can describe.
- $S \subseteq W$ is the subspace of *safe* states that are reachable without UB.

Under a natural monotonicity assumption on safe operations, we will show in Section 4.1 that S is upward closed and therefore open in the Alexandroff topology.

3.2 Simplicial Complexes and Homology

A *simplicial complex* K consists of a set of vertices V and a collection of finite subsets $\sigma \subseteq V$ called simplices, such that every subset of a simplex is also a simplex. We follow standard algebraic topology [1].

In our setting:

- 0-simplices represent states (vertices),
- 1-simplices represent safe transitions (edges),
- 2-simplices represent commuting squares of transitions (two independent operations that can be performed in either order),
- and so on.

For each integer $n \geq 0$, let $C_n(K)$ be the free abelian group on the set of n-simplices of K. We choose an orientation for each simplex. The boundary map

$$\partial_n : C_n(K) \to C_{n-1}(K)$$

is defined on a basis simplex $[v_0 \dots v_n]$ by

$$\partial_n [v_0 \dots v_n] = \sum_{i=0}^{n} (-1)^i [v_0 \dots \widehat{v_i} \dots v_n],$$

where $\widehat{v_i}$ means that the vertex v_i is omitted.

The nth homology group is

$$H_n(K) = \ker(\partial_n) / \operatorname{im}(\partial_{n+1}).$$

The group $H_1(K)$ detects loops that are not boundaries of any 2-simplex. The group $H_2(K)$ detects voids bounded by triangular faces, and so on.

3.3 Relative Homology

If $A \subseteq K$ is a subcomplex (for example a safe region) and K is a larger complex (for example including hypothetical or UB states), the relative chain group is

$$C_n(K, A) = C_n(K)/C_n(A).$$

The induced boundary maps $\partial_n^{\mathrm{rel}}$ give relative homology groups

$$H_n(K, A) = \ker(\partial_n^{\mathrm{rel}}) / \operatorname{im}(\partial_{n+1}^{\mathrm{rel}}).$$

Intuitively, $H_n(K, A)$ measures holes in K that do not already appear in A. In our setting this lets us isolate phenomena that only appear when we include unsafe states.

3.4 Category Theory: Lifetimes, Types and Reborrows

We recall the basic categorical notions that we need, following Mac Lane [5].

- A *category* $\mathcal{C}$ has objects and morphisms, with composition and identities.
- Any poset $(P, \leq)$ can be viewed as a *thin* category: there is at most one morphism $x \to y$, which exists iff $x \leq y$.
- A *functor* $F : \mathcal{C} \to \mathcal{D}$ maps objects and morphisms, preserving identities and composition.
- A *natural transformation* $\eta : F \Rightarrow G$ is a family of morphisms $\eta_c : F(c) \to G(c)$ such that for any $f : c \to c'$,
$$G(f) \circ \eta_c = \eta_{c'} \circ F(f).$$

We write:

- $\mathcal{L}$ for the category of *lifetimes* or origins. Its objects are lifetime variables and its morphisms are "outlives" relations.
- $\mathcal{T}$ for the category of *types*, with morphisms representing admissible coercions or subtypings.

A reference type such as Rust `&'a T` can be described as a bifunctor

$$\mathrm{Ref} : \mathcal{L}^{op} \times \mathcal{T} \to \mathcal{T}, \quad \mathrm{Ref}('a, T) = \&'aT,$$

which is covariant in T and contravariant in `'a`. Similar functors can model Mojo origin-indexed types [7, 8].

A reborrow operation (such as turning a mutable reference into a shared reference in Rust, or taking a span in Mojo that refers to the same origin) can be treated as a natural transformation between such functors. Naturality expresses that reborrows behave uniformly with respect to lifetime morphisms.

Remark on direction. The operational semantics of a program is inherently directed in time, while the simplicial complexes and homology groups that we use in this first treatment are undirected. In other words, homology only sees the underlying undirected shape of the graph of abstract worlds. This is deliberate: we focus here on the existence of non fillable cycles and cavities, not on the fine structure of temporal ordering. A more refined account could use notions of directed homotopy or directed homology, which explicitly encode the arrow of time, but we leave that extension for future work.

3.5 Sheaves on Posets and Cohomology (informal)

We briefly recall the basic notions that we rely on. Full details can be found in standard references on sheaves over posets, for example Hu [2].

Given a poset $(P, \leq)$ with the Alexandroff topology, the open sets are the upward closed subsets of P. A *presheaf* $\mathcal{F}$ on P assigns to each open set $U \subseteq P$ a set $\mathcal{F}(U)$ of "sections" over U, together with a restriction map

$$\rho_{U,V} : \mathcal{F}(U) \to \mathcal{F}(V)$$

for every inclusion of opens $V \subseteq U$, subject to the usual functoriality conditions

$$\rho_{U,U} = \mathrm{id}, \qquad \rho_{V,W} \circ \rho_{U,V} = \rho_{U,W} \text{ whenever } W \subseteq V \subseteq U.$$

Intuitively, $\mathcal{F}(U)$ collects all ways to assign data to the points of U, and restriction passes from larger regions to smaller ones.

A presheaf $\mathcal{F}$ is a *sheaf* when it satisfies the following locality and gluing property. If $\{U_i\}_{i \in I}$ is a cover of an open set U and we have a family of sections $s_i \in \mathcal{F}(U_i)$ that agree on pairwise intersections $U_i \cap U_j$, then there exists a unique global section $s \in \mathcal{F}(U)$ whose restriction to each U_i is s_i. Sheaves therefore capture when locally defined and pairwise compatible data can be glued into a single global object.

The cohomology groups $H^n(P, \mathcal{F})$ of a sheaf $\mathcal{F}$ over a poset P are algebraic invariants that measure the failure of such gluing to hold globally. In particular, the first cohomology group $H^1(P, \mathcal{F})$ can be interpreted as classifying obstructions to the existence of globally consistent sections built from locally defined ones. In the setting of this paper we will only need the informal idea that a nonzero class in H^1 signals that there is no single global assignment that matches all the local constraints, even though locally everything appears consistent.

4 The Geometric Framework

A *world* $w \in W$ is an abstract state that summarizes the information that is relevant for memory safety. We do not fix a single representation, but for concreteness one can think of w as a tuple

$$w = (Loc_w, Obj_w, Lft_w, Perm_w)$$

where:

- Loc_w is a finite set of abstract locations that represent heap or stack cells,
- Obj_w assigns to each location an abstract object and its type,
- Lft_w is a finite partial order of lifetime or origin tokens that tracks which objects or references outlive which others,
- $Perm_w$ assigns to each reference or pointer its current permission, for example read only, unique mutable, or invalid.

This structure is intentionally schematic. Different languages and different analyses can plug in their own abstract domains for objects, lifetimes and permissions. The only requirement for our framework is that there is a notion of *safe step* between worlds.

We write $w \leq w'$ when w' can be obtained from w by a finite sequence of such safe steps: that is, by allocations, borrows, reborrows, mutations and deallocations that do not violate the lifetime and permission discipline of the language or the chosen abstraction. This relation is reflexive and transitive, and in typical abstract domains antisymmetry holds up to the usual identification of abstract states, so $(W, \leq)$ is a poset.

4.1 Topology of Safe States

The *safe region* $S \subseteq W$ is defined as the set of worlds that are reachable from the initial world w_0 by performing only safe operations (under the language rules and the chosen abstraction).

Lemma 1. *If every safe operation is monotone with respect to $\leq$, in the sense that applying a safe operation to a world w yields a world w' with $w \leq w'$, then the set S of safe worlds is upward closed in $(W, \leq)$.*

Proof. Let $w \in S$ and suppose $w \leq w'$. By definition there is a path of safe operations from w_0 to w. The relation $w \leq w'$ means that there is a finite sequence of safe operations that extends w to w'. Concatenating the two sequences gives a path of safe operations from w_0 to w', so $w' \in S$. Hence S is upward closed. □

By the definition of the Alexandroff topology this means that S is an open subset of W.

We now build a simplicial complex K_S with:

- vertices given by worlds in S,
- edges representing single safe steps between worlds,
- higher-dimensional simplices representing commuting diagrams of safe transitions. For instance, a 2-simplex $[w_0, w_1, w_2]$ means that two different sequences of atomic steps from w_0 reach a common world w_2 and are considered equivalent.

We say that two operations a and b are *independent* in a given abstraction when they act on disjoint parts of the abstract state and their abstract transformers commute:

$$a(b(w)) = b(a(w))$$

for all worlds w in a certain region of interest. In a heap based abstraction this can be approximated by requiring that the sets of locations that a and b may read or write are disjoint. Whenever this independence condition holds and each individual step is safe, we add a 2 simplex to record that the two orders of execution are identified in the safe complex.

These higher simplices represent combinations of operations that do not interfere with each other.

4.2 UB as Holes

Consider a situation where two operations ought to be independent, but in fact one of the possible orders is invalid because of lifetimes or ownership. At the level of the complex:

- one edge or one face in the expected commuting diagram is missing,
- the boundary of the would-be diagram does not bound a full simplex in K_S.

The missing interior is a *hole*.

As a simple picture, suppose we have two operations op_1 and op_2 that we may try to apply to a world w_0.

$$\begin{tikzcd}
 & w_A \arrow[dr, "\mathsf{op}_2"] & \\
w_0 \arrow[ur, "\mathsf{op}_1"] \arrow[rr, "\mathsf{op}_2"'] & & w_B
\end{tikzcd}$$

If both orders (op_1 then op_2, and op_2 then op_1) are safe and lead to a common world, we can add a 2-cell that fills in this square. If one order is UB, that cell is missing and the boundary path yields a 1-cycle in K_S. In small examples such as the ones in Sections 5.1 and 5.2, this cycle represents a nontrivial element of $H_1(K_S)$, reflecting a one dimensional hole in the safe region.

Relative view. If we enlarge the complex to K_W, which includes all describable states (safe and unsafe), we obtain an inclusion $K_S \subseteq K_W$. The relative homology $H_n(K_W, K_S)$ then isolates the "unsafe cavities" that only appear when we allow UB states.

4.3 Categorical Overlay

We now add the categorical interpretation.

- Lifetimes and origins form a category $\mathcal{L}$.
- Types form a category $\mathcal{T}$.
- Reference types are bifunctors $\mathrm{Ref} : \mathcal{L}^{op} \times \mathcal{T} \to \mathcal{T}$.
- Variance is captured by functoriality in $\mathcal{L}$ and $\mathcal{T}$.
- Reborrows are natural transformations between these bifunctors, expressing coherence with lifetime morphisms.

UB often corresponds to the failure of such a naturality square. One may draw a square of lifetime changes and type coercions that ought to commute, but where no natural transformation makes everything fit. This is again analogous to a missing 2-cell in a diagram.

5 Applications

We now look at how this picture appears in concrete examples. For Rust we compute a small homology group explicitly. For C++ and Mojo we sketch the corresponding shapes.

5.1 Rust: Unsafe `as_ptr` and `Vec::push`

Consider the following Rust function.

```
fn ub(v: &mut Vec<i32>) -> i32 {
    let p = v.as_ptr();   // take raw pointer
    v.push(42);           // may reallocate, invalidating p
    unsafe { *p }         // UB if reallocation occurred
}
```

We ignore the element values and focus on lifetimes and borrow states for the vector buffer.

5.1.1 Abstract Worlds

Introduce the following abstract worlds.

- w_0: `v` exists and no raw pointer `p` has been taken.
- w_1: `p` is a raw pointer into the current buffer of `v`.

- w_2: the call `v.push(42)` has executed. The buffer may have moved, but we have not yet used `p`.
- w_3: `p` is dereferenced.

We distinguish safe transitions and the UB transition.

- Safe edges:
  - $e_{01} : w_0 \to w_1$ for the call `as_ptr`,
  - $e_{02} : w_0 \to w_2$ for the call `push` when no raw pointer exists yet.
- Problematic scenario:
  - going from w_1 to a state where `v.push(42)` has occurred and `p` is still assumed valid, and then dereferencing `p`. If the push reallocated, this is UB.

In a safe execution we either never take `p`, or we take it after the push, or we never dereference a possibly stale pointer. The unsafe order is what creates the hole.

5.1.2 Abstraction for the example

To connect the concrete Rust code to our worlds we sketch the abstraction that is applied to the function body. At the level of Rust MIR, the state of the function can be described by a stack of local variables together with the heap state. Our abstraction forgets most of that information and keeps only:

- whether the vector `v` is live and mutable,
- whether a raw pointer `p` into the buffer of `v` exists,
- whether the buffer of `v` has been potentially reallocated after `p` was taken.

The worlds w_0, w_1, w_2, w_3 introduced above are the images of the MIR states at the following program points:

- w_0 corresponds to the entry of the function, before the call to `as_ptr`,
- w_1 corresponds to the state just after `let p = v.as_ptr()`,
- w_2 corresponds to the state just after `v.push(42)` has executed,
- w_3 corresponds to the state at the unsafe dereference `*p`.

The unsafe pattern arises when execution follows the sequence $w_0 \to w_1 \to w_2 \to w_3$. If the push reallocated the buffer then, in a semantics such as Stacked Borrows, the pointer `p` no longer has a valid place on the borrow stack and dereferencing it is undefined. Our simplicial complex isolates the shape formed by the worlds that appear before the dereference, which is where the hole already appears.

5.1.3 A Minimal Simplicial Complex

To keep the homology calculation simple we collapse the situation to three vertices that reflect the essential shape.

- v_0: initial state, no pointer, mutable vector.
- v_1: pointer taken with `as_ptr`.
- v_2: vector mutated with `push`.

We draw the following diagram.

$$\begin{tikzcd}
& v_1 \arrow[dr, dashed, "\text{unsafe order}"] & \\
v_0 \arrow[ur, "\texttt{as\_ptr}"] \arrow[rr, "\texttt{push}"'] & & v_2
\end{tikzcd}$$

In a hypothetical world where `as_ptr` and `push` commute without UB, there would be a 2-simplex $\sigma = [v_0, v_1, v_2]$ whose boundary describes the three edge paths between these states. In the actual safe space, that simplex does not exist.

We work with a minimal toy complex K_{toy} given by the boundary of a 2-simplex:

- vertices v_0, v_1, v_2,
- edges $e_{01} = [v_0 v_1]$, $e_{12} = [v_1 v_2]$, $e_{20} = [v_2 v_0]$,
- no 2-simplices.

The chain groups are

$$C_1(K_S) \cong \mathbb{Z}^3, \quad C_0(K_S) \cong \mathbb{Z}^3, \quad C_2(K_S) = 0.$$

We orient the edges as follows.

- $e_{01} : v_0 \to v_1$,
- $e_{12} : v_1 \to v_2$,
- $e_{20} : v_2 \to v_0$.

The boundary map $\partial_1 : C_1 \to C_0$ is

$$\begin{aligned}
\partial_1(e_{01}) &= v_1 - v_0, \\
\partial_1(e_{12}) &= v_2 - v_1, \\
\partial_1(e_{20}) &= v_0 - v_2.
\end{aligned}$$

In matrix form (rows indexed by v_0, v_1, v_2, columns by e_{01}, e_{12}, e_{20}) we have

$$[\partial_1] = \begin{bmatrix} -1 & 0 & 1 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \end{bmatrix}.$$

Kernel of ∂_1. Write a general 1-chain as $x = ae_{01} + be_{12} + ce_{20}$. Then

$$\partial_1(x) = a(v_1 - v_0) + b(v_2 - v_1) + c(v_0 - v_2).$$

Collecting coefficients of each vertex gives

- coefficient of v_0 is $-a + c$,
- coefficient of v_1 is $a - b$,
- coefficient of v_2 is $b - c$.

The condition $\partial_1(x) = 0$ yields

$$-a + c = 0, \quad a - b = 0, \quad b - c = 0,$$

which forces $a = b = c$. Hence

$$\ker \partial_1 = \{a(e_{01} + e_{12} + e_{20}) \mid a \in \mathbb{Z}\} \cong \mathbb{Z}.$$

Image of ∂_2. Since $C_2(K_S) = 0$, the map ∂_2 is zero and $\operatorname{im}\partial_2 = 0$. Therefore

$$H_1(K_S) = \ker\partial_1 / \operatorname{im}\partial_2 \cong \mathbb{Z}.$$

We also have $H_0(K_S) \cong \mathbb{Z}$ (one connected component) and $H_n(K_S) = 0$ for all $n \geq 2$.

Interpretation. The generator of $H_1(K_S)$ is the loop

$$e_{01} + e_{12} + e_{20}.$$

This is a cycle that cannot be contracted within the safe region, because there is no 2-simplex filling in the interior. In words:

> There is a cycle of states (initial, pointer taken, vector mutated) such that the boundary of the expected commuting diagram is present, but the interior is missing.

The missing interior corresponds exactly to the forbidden interleaving where the raw pointer `p` is used after a possible reallocation. In a larger complex K_W that includes UB states one could add a 2-simplex representing this bad execution, which would kill the homology class in $H_1(K_W)$. The price is that this simplex has undefined or chaotic semantics.

This tiny complex is a minimal topological witness for the UB pattern that mixes `as_ptr` and `Vec::push`.

5.2 C++: Combined Raw Pointer and Iterator Invalidation

Consider the following C++ function, which combines two independent undefined behavior patterns in one place:

```
int g(std::vector<int>& v, std::string& s) {
    // assume v.size() >= 1 and s.size() >= 1
    int* p = &v[0];           // (1) raw pointer into v's buffer
    auto it = s.begin();      // (2) iterator into s

    v.push_back(42);          // (3) may reallocate, invalidating p
    s.erase(s.begin());       // (4) invalidates iterators, including it

    return *p + *it;          // (5) UB in both dimensions
}
```

Line (1) takes a raw pointer `p` into the buffer of `v`. Line (3) may reallocate the vector, invalidating `p`; dereferencing `p` afterwards is undefined. Similarly, line (2) takes an iterator `it` into `s`, line (4) erases the first character and invalidates `it`, and dereferencing `it` afterwards is undefined. These two hazards are logically independent.

We build an abstract state space that has one "cycle" for the vector pointer hazard and one for the string iterator hazard, sharing the same initial world. This yields a wedge of two circles and therefore a first homology group isomorphic to $\mathbb{Z}^2$.

5.2.1 Abstract worlds

We define the following abstract worlds, focusing only on the memory relevant shape:

- w_0: both `v` and `s` are alive and nonempty, but we have taken no pointer or iterator yet.
- w_{1v}: we have taken a raw pointer `p = &v[0]` into `v`; no iterator into `s`.
- w_{2v}: the buffer of `v` has been mutated in a way that may reallocate (for example via
- w_{1s}: we have taken an iterator `it = s.begin()` into `s`; no raw pointer into `v`.
- w_{2s}: the string `s` has been mutated in a way that invalidates iterators (for example via `s.erase(s.begin())`), and we no longer keep an iterator in scope.

The two hazards are represented as two triangles, each with the same initial world w_0 but with different intermediate worlds. We are not trying to model the entire control flow of `g` in this example. Instead, we isolate the local shapes around the two independent sources of undefined behavior.

5.2.2 Simplicial complex

We now define a simplicial complex $K_{\mathrm{C++}}$ that abstracts the local shape of these hazards; it is not intended to coincide exactly with the full safe complex K_S of the language, but rather to isolate the relevant cycles.

Vertices. The 0 simplices are the five abstract worlds

$$V = \{w_0, w_{1v}, w_{2v}, w_{1s}, w_{2s}\}.$$

Edges. We introduce six oriented edges (1 simplices), grouped into two triangles.

For the vector hazard:

- $e^v_{01} = [w_0 w_{1v}]$ represents taking the pointer `p = &v[0]`.
- $e^v_{12} = [w_{1v} w_{2v}]$ represents mutating `v` in a way that, in an idealized safe semantics, would coherently move us to a state where any outstanding pointer has been correctly accounted for.
- $e^v_{20} = [w_{2v} w_0]$ represents restoring an abstract state equivalent in shape to the initial one (for example by reinserting an element), again in an idealized view.

For the string hazard:

- $e^s_{01} = [w_0 w_{1s}]$ represents taking the iterator `it = s.begin()`.
- $e^s_{12} = [w_{1s} w_{2s}]$ represents mutating `s` in a way that, in an idealized safe semantics, would coherently account for the invalidation of iterators.
- $e^s_{20} = [w_{2s} w_0]$ represents restoring an abstract state equivalent to the initial one for `s`.

In reality, C++ does not give such a coherent semantics for invalidated pointers or iterators: the states that would lie on these edges either never arise or, if they do, any subsequent dereference is undefined. The complex $K_{\mathrm{C++}}$ therefore represents the boundary of two “ideal” commuting triangles, glued at the common world w_0.

Higher simplices. We do not include any 2 simplices. In particular, neither of the triangular loops

$$[w_0, w_{1v}, w_{2v}] \quad \text{or} \quad [w_0, w_{1s}, w_{2s}]$$

is filled in. The complex $K_{\mathrm{C++}}$ is therefore a wedge of two circles: two triangular loops attached at the shared vertex w_0.

5.2.3 Homology of the wedge of two triangles

We now compute the simplicial homology of $K_{\mathrm{C++}}$.

Chain groups. Let us order the vertices as

$$(w_0, w_{1v}, w_{2v}, w_{1s}, w_{2s}).$$

The 0 chain group is

$$C_0(K_{\mathrm{C++}}) \cong \mathbb{Z}^5,$$

freely generated by the five vertices.

The 1 chain group is generated by the six oriented edges:

$$C_1(K_{\mathrm{C++}}) \cong \mathbb{Z}^6,$$

with basis

$$(e^v_{01}, e^v_{12}, e^v_{20},\ e^s_{01}, e^s_{12}, e^s_{20}).$$

There are no 2 simplices, so $C_2(K_{\mathrm{C++}}) = 0$.

Boundary map $\partial_1 : C_1 \to C_0$. For an oriented edge $[xy]$ we have $\partial_1[xy] = y - x$. Thus:

$$\begin{aligned}
\partial_1(e^v_{01}) &= w_{1v} - w_0,\\
\partial_1(e^v_{12}) &= w_{2v} - w_{1v},\\
\partial_1(e^v_{20}) &= w_0 - w_{2v},\\
\partial_1(e^s_{01}) &= w_{1s} - w_0,\\
\partial_1(e^s_{12}) &= w_{2s} - w_{1s},\\
\partial_1(e^s_{20}) &= w_0 - w_{2s}.
\end{aligned}$$

In matrix form, with rows indexed by $(w_0, w_{1v}, w_{2v}, w_{1s}, w_{2s})$ and columns by $(e^v_{01}, e^v_{12}, e^v_{20}, e^s_{01}, e^s_{12}, e^s_{20})$, we have

$$[\partial_1] = \begin{bmatrix} -1 & 0 & 1 & -1 & 0 & 1\\ 1 & -1 & 0 & 0 & 0 & 0\\ 0 & 1 & -1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & -1 & 0\\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}.$$

Kernel of ∂_1. A general 1 chain is

$$x = a_1 e^v_{01} + a_2 e^v_{12} + a_3 e^v_{20} + b_1 e^s_{01} + b_2 e^s_{12} + b_3 e^s_{20},$$

with $a_i, b_i \in \mathbb{Z}$. Applying ∂_1 and collecting coefficients gives the system

$$\begin{aligned}
\text{at } w_0 &: -a_1 + a_3 - b_1 + b_3 = 0,\\
\text{at } w_{1v} &: a_1 - a_2 = 0,\\
\text{at } w_{2v} &: a_2 - a_3 = 0,\\
\text{at } w_{1s} &: b_1 - b_2 = 0,\\
\text{at } w_{2s} &: b_2 - b_3 = 0.
\end{aligned}$$

From the last four equations we obtain

$$a_1 = a_2 = a_3 =: a, \qquad b_1 = b_2 = b_3 =: b$$

for some $a, b \in \mathbb{Z}$. Substituting into the first equation gives

$$(-a) + a + (-b) + b = 0,$$

which is identically satisfied for all a, b. Thus every cycle has the form

$$x = a(e^v_{01} + e^v_{12} + e^v_{20}) + b(e^s_{01} + e^s_{12} + e^s_{20}).$$

It follows that

$$\ker \partial_1 = \{a\ell_v + b\ell_s \mid a, b \in \mathbb{Z}\} \cong \mathbb{Z}^2,$$

where we have defined the two fundamental loops

$$\ell_v = e^v_{01} + e^v_{12} + e^v_{20}, \qquad \ell_s = e^s_{01} + e^s_{12} + e^s_{20}.$$

Image of ∂_2. There are no 2 simplices in $K_{\mathrm{C++}}$, so $C_2 = 0$ and $\partial_2 : C_2 \to C_1$ is the zero map. Hence

$$\operatorname{im} \partial_2 = 0.$$

Homology groups. The first homology group is

$$H_1(K_{\mathrm{C++}}) = \ker \partial_1 / \operatorname{im} \partial_2 \cong \ker \partial_1 \cong \mathbb{Z}^2.$$

The two generators can be chosen as the homology classes of ℓ_v and ℓ_s, which correspond to the two independent undefined behavior patterns: one for the raw pointer into `v` and one for the iterator into `s`.

The zeroth homology group H_0 measures connected components. The complex $K_{\mathrm{C++}}$ is connected (all vertices are reachable from w_0 by edges), so $H_0(K_{\mathrm{C++}}) \cong \mathbb{Z}$. For $n \geq 2$ there are no n simplices, so $H_n(K_{\mathrm{C++}}) = 0$.

5.2.4 Interpretation

Compared to the Rust and single hazard examples, which yield a single independent loop and $H_1 \cong \mathbb{Z}$, this C++ construction yields a wedge of two loops and $H_1 \cong \mathbb{Z}^2$. Each generator corresponds to a distinct, structurally independent undefined behavior cavity in the abstract state space: one for raw pointers invalidated by vector reallocation, and one for iterators invalidated by string erasure. In a larger complex that included states with dangling pointers and iterators, these loops could be filled by 2 simplices representing executions that actually pass through undefined behavior. Restricting attention to the safe region means that the corresponding 2 simplices are absent, and their absence is reflected algebraically by the nontrivial first homology group.

5.3 Mojo: Origins and Origin Unions

Mojo has explicit origin tracking for values [7, 8, 9]. Informally:

- references such as `Pointer[T,` α`]` or `Span[T,` β`]` carry origins α, β,
- origin unions such as $\alpha \vee \beta$ represent values that may have come from origin α or origin β.

For our purposes a Mojo world contains:

- a set O_w of live origins with a partial order that reflects nested scopes and lifetime relationships, and
- a set of references whose origins belong to O_w.

We say $w \leq w'$ when w' extends lifetimes or adds origins in a way that respects Mojo's rules, that is, no reference outlives the origins it depends on.

The origin poset $(O, \leq)$ can itself be treated as a small category. Reference types are functors of the form

$$\mathrm{Ptr} : O^{op} \times \mathcal{T} \to \mathcal{T},$$

and origin unions give derived origins in O. An origin union $\gamma = \alpha \vee \beta$ is interpreted conservatively: a value with origin γ may have come from either α or β, so the static rules require that any use of such a value be valid regardless of which source was chosen. Operationally this means that γ is constrained to respect *both* α and β rather than being tied to just one of them; in terms of the lifetime order this is enforced by requiring appropriate inequalities between γ, α and β along all control flow paths.

A simple undefined behavior pattern in Mojo looks similar to the Rust and C++ examples:

- create a `Span[T, `α`]` from an array with origin α,
- let α expire when that array goes out of scope,
- keep using the `Span`.

The shape of the abstract state space is the same as before. There is a path from "array alive, no span" to "span exists" and a path from "array alive, no span" to "array destroyed", but there is no safe interior region that combines both "span live" and "origin expired". The missing interior is again a one dimensional hole in the safe region, which in a larger complex K_W would be filled by an undefined behavior state where the span points into dead storage.

Origin unions introduce another kind of global consistency condition. Suppose we have origins α and β, and we form a union $\gamma = \alpha \vee \beta$. Mojo's typing rules treat γ as a value that could have come from either α or β, and enforce constraints that ensure any use of γ is safe under both cases. Locally, along a particular control flow path, these constraints may appear consistent: every use of γ lies in a region where both α and β are still live according to the analysis. Globally, however, different regions of the program may impose different inequalities between α, β and γ, and it may turn out that there is no single assignment of origins and lifetimes that satisfies all of them at once.

This suggests that cohomological invariants on a graph or poset of origins can be useful, in the sense sketched in Section 3.5. One can model the flow of origin information as a presheaf on a poset of program regions and consider sheaf cohomology on the corresponding Alexandroff space [2]. A nonzero class in H^1 with coefficients in an appropriate abelian group expresses the fact that there is no global potential function that is compatible with all the local constraints. In our setting such a class would correspond to a pattern of origin unions and uses that cannot be given a globally consistent lifetime assignment, even though the local checks along each region may look acceptable. We do not develop this formalization here, but we view it as a natural next step.

6 Homology Based UB Detection: A High Level Algorithm

We now outline how the geometric construction can be used inside a static analyzer or research prototype. The procedure is intentionally high level and is meant to match the kind of pipeline that a modern compiler or analysis tool could implement.

Algorithm: DetectUBHoles

```
Require: Function or basic block $F$ in Rust, C++ or Mojo
Ensure: Set $\mathcal{H}$ of candidate UB holes, each annotated with a loop of abstract states and the
    corresponding program operations
 1: $\mathcal{H} \leftarrow \emptyset$
--------------------------------------------------------------------------------
    Phase 1: Build abstract worlds and transitions
 2: Build the control flow graph (CFG) of $F$
 3: for each program point $p$ in the CFG do
 4:     Compute an abstract world $w_p$ that records:
          live objects
          lifetimes or origins
          references or pointers and permissions
 5: end for
 6: $W_F \leftarrow \{w_p \mid p$ is a program point in $F\}$
--------------------------------------------------------------------------------
    Phase 2: Construct the safe 1 skeleton
 7: Initialize a directed graph $G$ with vertex set $W_F$
 8: for each CFG edge $(p \to q)$ do
 9:     if the step from $w_p$ to $w_q$ is not provably UB under the abstraction then
10:         Add a directed edge $w_p \to w_q$ to $G$
11:     end if
12: end for
--------------------------------------------------------------------------------
    Phase 3: Add commuting 2 cells
13: Initialize a simplicial complex $K_S$ whose vertices are the nodes of $G$
14: Add a 1 simplex to $K_S$ for each edge of $G$
15: for each pair of independent operations that may commute do
16:     Look for squares of the form:
          $w \xrightarrow{a} w_1$, $w \xrightarrow{b} w_2$
          $w_1 \xrightarrow{b'} w_{12}$, $w_2 \xrightarrow{a'} w_{21}$
17:     if $w_{12} = w_{21}$ and all four edges are safe then
18:         Add the corresponding 2 simplex (or family of 2 simplices) to $K_S$
19:     end if
20: end for
--------------------------------------------------------------------------------
    Phase 4: Optional larger complex
21: Optionally construct a larger complex $K_W$ by adding:
          vertices for hypothetical UB worlds
          edges for UB transitions
          simplices that complete commuting diagrams if UB is allowed
--------------------------------------------------------------------------------
    Phase 5: Compute homology
```

```
22: Compute boundary matrices $\partial_1$ and $\partial_2$ for $K_S$
23: Compute a basis for $H_1(K_S) = \ker(\partial_1)/\operatorname{im}(\partial_2)$
24: if $K_W$ was constructed then
25:     Compute the relative boundary maps for $(K_W, K_S)$
26:     Optionally compute a basis for $H_2(K_W, K_S)$
27: end if
```

Phase 6: Interpret homology generators

```
28: for each generator $z$ of $H_1(K_S)$ do
29:     Map $z$ to a loop of vertices and edges in $G$
30:     Recover the corresponding program points and operations
31:     Add a candidate UB report to $\mathcal{H}$ containing:
          the loop in world space
          the sequence of source locations and operations
32: end for
```

Phase 7: Heuristics and pruning

```
33: Optionally restrict the analysis to:
        a single function, loop body or basic block
        regions around unsafe code or suspicious patterns
34: Optionally refine the abstraction only in regions where nontrivial homology was detected
35: Mark the resulting set $\mathcal{H}$ as the collection of candidate UB holes to report
36: return $\mathcal{H}$
```

The output of the DetectUBHoles algorithm is a set of candidate UB patterns that correspond to nontrivial homology classes. Each candidate is backed by a loop in the abstract state space and by a concrete sequence of operations in the source code. Existing static analyses can supply the abstract worlds and safe steps, and the homology computation follows standard linear algebra over the chain complexes of the simplicial complex.

From a computational point of view, if there are N abstract worlds and M safe edges in the region under analysis, then the 1 skeleton of K_S has N vertices and at most M edges, and the number of 2 simplices is bounded by the number of commuting squares that the analysis identifies. In an intraprocedural setting these numbers are typically modest. Standard algorithms for computing H_1 of a finite simplicial complex reduce to linear algebra over $\mathbb{Z}$ or over a finite field and are cubic in the size of the boundary matrices in the worst case, but are quite practical for the small complexes produced by a per function or per loop analysis. The overall workflow would augment an existing compiler pipeline: after borrow checking or a similar static analysis has produced abstract states and safe transitions, the homology based phase would run on selected regions (for example around unsafe blocks or suspicious pointer patterns) and report any nontrivial cycles as candidate UB holes.

7 Related Work

Formal semantics for systems languages. There is a substantial body of work on formal models of memory safety and UB in systems languages. RustBelt [4] and Oxide [10] give logical relations based semantics for Rust, and Stacked Borrows [3] refines the aliasing model for raw pointers and references. Cerberus [6] gives a parameterized semantics for C that exposes UB in a modular way. These approaches provide deep guarantees about well typed programs but do not describe the global shape of the state space in topological terms.

Static and dynamic tools. On the tooling side, there are many static analyzers and dynamic sanitizers for C and C++ that aim to detect UB patterns, such as use after free and invalidated iterators. Rust's borrow checker plays a similar role at the level of the type system, while tools based on abstract interpretation and dataflow analysis operate on lower level intermediate representations. Our homology based view is not a replacement for these techniques, but rather a conceptual layer that can sit on top of them: it uses the abstract states and transitions they provide to build a complex whose holes signal structural problems.

Topological and categorical methods. Topology and category theory have been used in the study of programs in several ways, including domain theory, game semantics and higher categorical models of type theory. In concurrency, directed homotopy and higher dimensional automata are used to capture the space of executions. Closer to our setting, there is work on sheaves over posets and their cohomology [2], which provides tools for reasoning about local to global consistency. Our contribution is to bring simplicial homology and relative homology into the discussion of UB and lifetime semantics in concrete systems languages, and to suggest that holes in the resulting complexes capture patterns that traditional analyses identify only indirectly.

8 Limitations

It is important to be clear about the limits of the present work. First, the framework is defined at the level of an abstraction of program states. Different abstractions may expose or hide different shapes, and there is no guarantee that every instance of UB in the concrete semantics corresponds to a nontrivial homology class in the chosen abstract complex. Second, we do not prove that the absence of holes implies the absence of UB, even at the abstract level. The homology based analysis is intended as a bug finding heuristic and a conceptual lens, not as a complete verification technique. Third, the simple complexes that we construct in the examples are intraprocedural and single threaded. Whole program analysis, foreign function interfaces and concurrency introduce additional structure that would require richer notions of state and more sophisticated topological tools. Finally, our discussion of categorical and sheaf theoretic ideas is deliberately high level. Developing those aspects into full formal models is part of the future work outlined below.

9 Future Work

This framework suggests several directions for further work.

- **Prototype tools.** One can build a prototype that consumes Rust MIR or a C++ intermediate representation and constructs small complexes around uses of unsafe features. The tool can then check H_1 locally and compare the results to reports from sanitizers or to known UB bugs.
- **Type system integration.** Homological properties can serve as meta-invariants. For example, type rules could be designed in such a way that they guarantee that certain classes of complexes are contractible, which would imply the absence of specific UB patterns.

- **Concurrency and directed topology.** Extending worlds with thread interleavings and synchronization information would lead to higher-dimensional spaces. Techniques from directed homotopy and directed homology could help detect data races and deadlocks as higher-dimensional holes.
- **Region-based and arena allocation.** Many systems use region or arena allocators. By treating regions as coordinates in the world space one can examine whether cross-region references introduce cycles in a region dependency graph that suggest UB.
- **Sheaf-theoretic invariants.** It is natural to assign to each open set of worlds a set of invariants (types, logical properties) and to regard these assignments as sheaves on a poset with the Alexandroff topology. Cohomology of such sheaves on posets [2] can reveal global inconsistencies in the same way that it does in other geometric settings.

Viewing UB as a geometric phenomenon, namely as holes in a structured state space, gives a language independent perspective on safety. It provides algebraic invariants that can complement more traditional type and logic based approaches and that may eventually guide both the design of languages and the construction of practical analysis tools.

10 Declaration of generative AI and AI assisted technologies in the writing process

The author used ChatGPT 5.1 (OpenAI) as an assistant during the preparation of this manuscript. ChatGPT 5.1 was used to help clarify and assess the original ideas, to support ideation and alternative formulations, and to generate early draft passages of text that were then heavily revised. It was also used for editing and refinement during later stages of the manuscript.

The most substantial specific pointers obtained from ChatGPT 5.1 were the suggestion to use the Alexandroff topology on the poset of abstract worlds and the identification of Hu's note on sheaf structures on posets [2] as a relevant reference. All AI generated text was reviewed and edited by the author, who takes full responsibility for the final content and for any remaining errors or omissions. No generative AI tools were used for data analysis, formal proofs, or experiments beyond the roles described above.

References

[1] Allen Hatcher. *Algebraic Topology.* Cambridge University Press, Cambridge, 2002.

[2] Chen-Shou Hu. A brief note for sheaf structures on posets. *arXiv preprint*, arXiv:2010.09651, 2020.

[3] Ralf Jung, Hoang-Hai Dang, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2020.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018.

[5] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer, New York, 2 edition, 1998.

[6] Kayvan Memarian. The cerberus c semantics. Technical Report UCAM-CL-TR-981, University of Cambridge, Computer Laboratory, 2023.

[7] Modular, Inc. Lifetimes, origins, and references. `https://docs.modular.com/mojo/manual/values/lifetimes`, 2025. Accessed: 2025-11-13.

[8] Modular, Inc. Pointers. `https://docs.modular.com/mojo/manual/pointers/`, 2025. Accessed: 2025-11-13.

[9] Modular, Inc. Value lifecycle. `https://docs.modular.com/mojo/manual/lifecycle/`, 2025. Accessed: 2025-11-13.

[10] Aaron Weiss, Daniel Patterson, and Amal Ahmed. Oxide: The essence of Rust. *arXiv preprint*, arXiv:1903.00982, 2019.