On Rust and Categorical Semantics

by Alejandro José Soto Franco

Last updated: July 29, 2025

Hosted on Github Manuscripts Repo · `WIP v1.3 Book`

Contents

Preface. This is a technical collection of computational and mathematical notes. No claims are made beyond what is written. The work is presented as is. As a work-in-progress draft, what is not yet rigorous may become so later. The author has tried to write only what he can verify. Please excuse inconsistencies in notation. These will be ironed out before final publication.

I High-Level Overview

At its core, Rust enforces a set of invariants corresponding to ownership, borrowing, aliasing, and lifetimes that allow memory-safe, data-race-free programs to be written *without a garbage collector*. Indeed, they are categorical constraints that form a coherent structure over program state. This structure demands a semantics that can:

1. Reflect fine-grained distinctions between move, copy, and borrow,
2. Track the availability and uniqueness of morphisms over program variables,
3. Interpret execution as a homotopically meaningful path through a space of typed states.

Traditional denotational or operational semantics fails to adequately model these distinctions without either overcomplicating the syntax or obscuring the key invariants. Instead, we adopt a semantics based on:

- **Affine type theory**: where variables are resources that may be consumed at most once,
- **Sheaf-theoretic execution models**: where program state is local data with gluing constraints over time,
- **Category-theoretic morphisms**: where ownership and borrow rules translate directly to uniqueness and commutativity of morphisms in a type-preserving category.

This perspective allows us to formalize Rust's ownership system as a coherent semantic geometry of execution: a system of type-preserving, resource-indexed transformations over program state.
Each of Rust's core safety rules corresponds to a precise invariant in this execution geometry, the logic underlying which we will explore in the course of this text:

Mutable access	$\rightsquigarrow$	A unique morphism $f : x \to x'$ over an affine resource x, disallowing parallel branches,
Shared access	$\rightsquigarrow$	A family of commuting morphisms $\{f_i : x \to x\}_{i \in I}$, preserving read-only semantics,
Lifetimes	$\rightsquigarrow$	Restriction of morphism support to a region $U \subseteq \mathcal{T}$, where $\mathcal{T}$ is program time,
Move semantics	$\rightsquigarrow$	Affine context contraction $\Gamma \vdash e : \tau \Longrightarrow \Gamma \setminus \{x\} \vdash e' : \tau'$, consuming x.

By treating these semantics as the *primary object of study*, we are able to:

- Define well-formed execution paths as categorical compositions,
- Make precise statements about soundness, race-freedom, and memory correctness,
- Generalize Rust's ideas to new safe languages or embedded DSLs with provable safety guarantees.

Ultimately, Rust offers a bridge between high-performance computation and formal reasoning. To cross it rigorously, we must treat the semantics not as a byproduct, but as a guiding structure. The goal of this document is to expose that structure — to show that Rust is not merely a language with rules, but a *geometry of execution* governed by types, and to construct the categorical semantics that makes this geometry coherent, composable, and verifiable.
The following are tables roughly corresponding to the topics discussed in each section:

Core Rust Constructs and Affine Semantics

Rust Construct	Semantic Interpretation
`let x = 5;`	Immutable binding. Disallows duplication (affine logic).
`let mut x = 5;`	Mutable binding. Enables linear update; weakening allowed.
`fn f(x: T) -> U`	Morphism $f : T \to U$ in affine type category.
`let b = a;`	Ownership move. Consumes 'a' unless 'T: Copy'.
`&a`	Shared borrow. Modal access $\Box T$; comonadic read-only.
`&mut a`	Exclusive borrow. Linear access $\triangleright T$; no aliasing.

Structural Rules of Affine Logic

Rule	Affine Logic	Rust Behavior
Weakening	Unused bindings permitted	Declared vars may go unused
Contraction	Disallowed	No duplication unless `Copy`
Exchange	Permitted	Statement order flexible

Ownership, Copy, and Borrowing

Operation	Condition	Semantic Class
Move	Default	Consumes value (linear logic)
Copy	If `T: Copy`	Duplication allowed; contraction admitted
Immutable borrow	Always	Comonadic: $\Box T$
Mutable borrow	Exclusive	Linear ephemeral: $\triangleright T$
Shadowing	Type-safe	New binding masks prior

Categorical Interpretation of Rust

Rust Concept	Categorical Structure
Types `T`	Objects in affine symmetric monoidal category AffType
Functions `fn`	Morphisms $f : T \to U$
Ownership Move	Consuming morphism (non-duplicative)
Copy	Identity-preserving duplication morphism
Borrowing	Modal morphisms $(\Box, \triangleright)$
Traits	Predicate subobjects or classifier constraints

Borrow Checker as Categorical Constraint

Invariant	Categorical or Logical Justification
No `&mut` if `&` exists	Morphism uniqueness in affine category
Only one `&mut` at a time	Linear typing discipline
Lifetimes	Indexed morphisms over regions; natural transformations
Reborrow	Subobject relation under scope restriction

Trait Semantics and Resource Flow

Trait	Purpose	Semantic Role
`Copy`	Implicit duplication	Contraction permitted
`Clone`	Explicit duplication	Manual morphism invocation
`Send`	Thread-safe transfer	Safe morphism across threads
`Sync`	Shared thread access	Shared morphism closure
`Drop`	Cleanup on scope end	Terminal morphism (finalizer)

Modal Typing Summary

Symbol	Rust Meaning	Typing Semantics
$\Diamond T$	`&T`	Comonadic, shared read-only
$\triangleright T$	`&mut T`	Linear, exclusive use
T	Owned value	Affine object (non-duplicable)

This perspective lets us model Rust's ownership system as a *semantic geometry of execution*: a space of type-preserving, morphism-governed transitions over program state. Each of Rust's core safety rules corresponds to a geometric invariant:

$$\begin{aligned}
\textbf{Mutable access} &\rightsquigarrow \text{A unique morphism } f : x \to x' \text{ over an affine resource } x, \text{ disallowing branching,}\\
\textbf{Shared access} &\rightsquigarrow \text{A commuting family } \{f_i : x \to x\}_{i \in I}, \text{ enabling read-only access,}\\
\textbf{Lifetimes} &\rightsquigarrow \text{Morphisms with support restricted to regions } U \subseteq \mathcal{T}, \text{ where } \mathcal{T} \text{ is program time,}\\
\textbf{Move semantics} &\rightsquigarrow \text{Affine context contraction } \Gamma \vdash e : \tau \Longrightarrow \Gamma \setminus \{x\} \vdash e' : \tau', \text{ consuming } x.
\end{aligned}$$

By centering these constraints semantically, we can:

- Define well-formed executions as categorical compositions,
- Prove properties like race-freedom and memory soundness,
- Generalize Rust's model to new safe languages or DSLs.

Ultimately, Rust offers a bridge between high-performance computing and formal semantics. To cross it rigorously, we must not treat semantics as a byproduct of syntax, but as its guiding structure. This document exposes that structure — showing Rust is not merely a language with rules, but a *geometry of execution* governed by affine types, modal morphisms, and coherent lifetimes.

II Preliminaries

This section introduces core syntactic and semantic elements of the `Rust` programming language that will recur throughout our mathematical treatment. We begin with primitive types, basic function definition, and ownership semantics. These elements will later be given categorical and geometric interpretations.

Definition II.1. Primitive Types in Rust.
The set of primitive types in `Rust` includes the fixed-size, compile-time known scalar types listed below. These types are stratified by logical, integer, and floating-point structure, each with a specified bit-width and memory layout.

Type	Bits	Signedness	Structure
`bool`	1	n/a	Logical type
`char`	32	Unicode scalar	Character code point
`u8`	8	Unsigned	Integer
`u16`	16	Unsigned	Integer
`u32`	32	Unsigned	Integer
`u64`	64	Unsigned	Integer
`usize`	Platform-dependent	Unsigned	Pointer-sized Integer
`i8`	8	Signed	Integer
`i16`	16	Signed	Integer
`i32`	32	Signed	Integer
`i64`	64	Signed	Integer
`isize`	Platform-dependent	Signed	Pointer-sized Integer
`f32`	32	IEEE 754	Floating-point
`f64`	64	IEEE 754	Floating-point

Types such as `u32`, `i64`, and `f64` form the basis of numeric computation, with arithmetic operators and ordering defined over them. The type `bool` encodes the two-valued logical space $\{\texttt{true}, \texttt{false}\}$, and `char` denotes valid Unicode scalar values, not necessarily ASCII. The types `usize` and `isize` track platform-dependent word width and are used for memory indexing. ⌟

Remark 1. (Type Safety as Categorical Discipline)
The type system of `Rust` is nominal, statically checked, and strongly typed. Every value in a well-formed program is assigned a single, unambiguous type that is fully determined at compile time. Type annotations, inference, and trait resolution collectively ensure that no value can be interpreted at runtime in a way inconsistent with its static type.
From a semantic perspective, this corresponds to a categorical discipline in which:

- *Types* are treated as objects in a category $\mathcal{C}_{\texttt{Rust}}$,
- *Expressions* correspond to morphisms $f : \tau_1 \to \tau_2$,
- *Well-typedness* enforces that all morphisms are defined only on valid objects, with respect to ownership, borrowing, and lifetimes,

- *Type errors* reflect *undefined morphisms*, and are excluded from the semantics entirely.

Thus, the typing judgment

$$\Gamma \vdash e : \tau$$

may be interpreted as asserting the existence of a partial morphism

$$f_e : \mathrm{Dom}_\Gamma \dashrightarrow \tau$$

defined only under the constraints enforced by the context Γ. This partiality is not due to non-termination or dynamic checks, but arises from the *logical geometry* of affine resources and the static structure of the program.

Categorically, the restriction to well-typed programs corresponds to working in a *typed subcategory* $\mathcal{C}_{\text{safe}} \subseteq \mathcal{C}_{\texttt{Rust}}$, where morphisms preserve not only type structure but also aliasing discipline and memory soundness. As such, type safety in `Rust` is not merely a compiler check — it is the formal criterion by which the composability and realizability of morphisms in the execution space is guaranteed. We will later see that lifetimes impose sheaf-theoretic support conditions, and ownership corresponds to affine contraction laws, all of which are enforced through this categorical structure. ⟼

Definition II.2. Functions in Rust.
A function is declared via the syntax:

```
fn f(x: i32) -> i32 {
    x * x
}
```

This defines a mapping $f : \mathbb{Z} \to \mathbb{Z}$ via the rule $x \mapsto x^2$. The compiler checks that all branches return the declared type. ⌟

Problem 1. (Evens and Odds)
Write a function in `Rust` that determines whether an integer is even. ⌟

Solution.

```
fn is_even(n: i32) -> bool {
    n % 2 == 0
}
```

This uses modulo arithmetic and returns a `bool`.

Definition II.3. Immutability and Mutability.
In `Rust`, variables are immutable by default:

```
let x = 5;          // x is immutable
let mut y = 5;      // y is mutable
```

This immutability reflects an affine logic perspective. In later sections we will interpret this as the default denial of contraction. ⌟

Example. In contrast to many languages, `Rust` enforces that mutation must be explicit. The ownership and borrowing rules ensure safe concurrency and eliminate data races by construction. ♢

Definition II.4. Ownership.
Each value in `Rust` has a single owner at any time. When ownership is transferred (moved), the previous binding becomes invalid:

```
let s1 = String::from("hello");
let s2 = s1; // ownership moved to s2
// s1 is no longer valid
```

This model ensures no two variables alias the same memory without the borrow checker knowing. ⌟

Remark 2. (Ownership as Morphism Restriction)
`Rust`'s ownership system enforces a precise constraint on how program state evolves: each value has a unique owner, and this ownership must be explicitly transferred, borrowed, or consumed. From a semantic perspective, this corresponds to a structural constraint that determines which morphisms are defined between typed program states, with execution valid only when transitions preserve ownership.
Let $\mathcal{C}$ be a category whose objects are well-typed program states, and whose morphisms represent valid transitions between these states induced by program expressions. In this setting, *ownership* corresponds to a condition on morphism formation: a morphism $f : x \to y$ is defined only when the value x has not been aliased, borrowed mutably, or consumed elsewhere in the execution trace. The morphism f must *consume* x from the domain, reflecting that the resource is no longer available thereafter.
Formally, we express this as a *partial morphism restriction*:

$$\mathrm{Hom}_{\mathrm{own}}(x, y) \subseteq \mathrm{Hom}_{\mathcal{C}}(x, y)$$

where $\mathrm{Hom}_{\mathrm{own}}(x, y)$ denotes morphisms respecting ownership constraints. This substructure of the full morphism space captures the affine typing discipline: each resource (variable binding) may be used at most once in constructing morphisms, unless the type supports duplication (e.g., via 'Copy').
This restriction induces a geometric structure on the manifold of program execution: not all paths between states are valid, only those corresponding to ownership-preserving morphisms. Executing a program then becomes equivalent to tracing a directed path through this manifold, where each step consumes inputs in accordance with affine type constraints and produces new, uniquely owned outputs.
In later sections, we will extend this picture to include borrow semantics as morphisms into read-only fibers, and lifetimes as region-restricted morphism supports. Ownership, in this unified model, is not a language feature but a manifestation of a deeper constraint: *that morphisms over state must respect the affine geometry of consumption and exclusivity.* ⤚

Definition II.5. Borrowing.
Rust allows references to values without transferring ownership:

```
fn len(s: &String) -> usize {
    s.len()
```

}

Here `&String` is an immutable reference. Mutable references `&mut T` are exclusive and non-aliasable. ⌟

Problem 2. (Borrow Checker Conflict)
Explain why the following Rust code results in a compile-time error:

```
let mut x = 5;
let y = &x;
let z = &mut x;
```

⌟

Solution. The error is due to a violation of Rust's **exclusive aliasing invariant**, a core enforcement of its **affine type system**. This rule is rooted in affine logic, which permits weakening (discarding unused resources) but disallows contraction (duplicating them). Rust translates this into the following runtime constraint:

- You may have either one mutable reference (`&mut T`) **or**
- Any number of immutable references (`&T`),
- **But not both simultaneously**.

In the given code:

- `x` is mutable and is first immutably borrowed as `y : &i32`,
- Then `z : &mut i32` is created while `y` is still in scope,

violating Rust's borrowing rules. From the perspective of type systems:

- Immutable borrows preserve resource identity and do not consume it.
- Mutable borrows **consume** the ability to alias or share, requiring exclusive access.

In categorical semantics, this behavior can be modeled in an affine symmetric monoidal category:

- Let $\mathsf{AffType}$ be a category of affine types,
- A value of type T is an object,
- A mutable borrow is a morphism $\mathsf{mut\_ref}_T : T \to \&\mathsf{mut}\, T$ that consumes T,
- An immutable borrow is a morphism $\mathsf{shared\_ref}_T : T \to \Box T$ where $\Box$ is a comonadic modality preserving read-only access.

The conflict arises because you attempt to create `z` while `y` still exists. This would force a diagram with both `&T` and `&mut T` arrows in the same scope, violating uniqueness of morphisms in the affine category. **Thus, the compiler prevents this construct at compile-time to uphold memory safety and preserve the semantics of linear resource flow.**

Remark 3. (Aliasing Rule and Morphism Uniqueness)
This restriction prevents data races and enables non-interleaved reasoning about program state. From a semantics perspective, we will later encode this as a uniqueness condition on morphisms.
To formalize this, recall that in Rust, at any given time, one of the following must hold for a variable binding:

(i) Exactly one mutable reference exists, and no immutable references,

(ii) Any number of immutable references exist, and no mutable references.

We interpret the program heap (or semantic memory) as a fibered category of stateful objects. Let $\mathcal{C}$ be the category of memory configurations, and let each morphism $f : x \to y$ represent a permitted transformation of state via a function or method application.

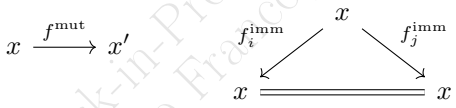
Then, the *aliasing rule* corresponds to the requirement that:

$$\forall x \in \mathrm{Ob}(\mathcal{C}), \quad \exists!\, f_x^{\mathrm{mut}} : x \to x' \quad \text{or} \quad \forall i, f_i^{\mathrm{imm}} : x \to x \quad \text{such that } [f_i, f_j] = 0 \ \forall i, j. \tag{1}$$

That is, either:

- There exists a unique morphism representing mutable access (exclusive right to evolve the object), or
- There exists a commuting family of morphisms representing immutable access (read-only projections).

This ensures the state evolution diagram remains non-branching under mutation and commutative under observation:



Categorically, this induces a *uniqueness constraint* on the morphisms permitted within a given context. We can write this as a context morphism condition:

$$\Gamma \vdash \exists!\, f : x \to y \quad \text{or} \quad \Gamma \vdash \{f_i : x \to x\}_{i \in I} \text{ with } [f_i, f_j] = 0. \tag{2}$$

This encoding provides a semantic justification for Rust's aliasing rules, and also serves as a foundational rule for safe, concurrent execution. Uniqueness of morphisms corresponds to thread safety guarantees and compositional determinism.

This rule will later appear as a coherence law in our Ω-type model of Rust's ownership system as part of the categorical semantics framework we seek to build across all possible typed geometric systems, including Clifford-algebraic, cohomological, and modal resource-sensitive logics. ⟶

Definition II.6. Shadowing.
Rust allows a variable to be re-bound with the same name:

```
let x = 5;
let x = x + 1;
```

The second `x` shadows the first. This differs from mutation and does not require `mut`. ⌟

Example. Shadowing is useful for type transformations:

```
let spaces = "   ";
let spaces = spaces.len(); // spaces is now a usize
```

i.e., a `&str` becomes a `usize`. ♢

Proposition II.7. Shadowing's Usefulness.
Shadowing introduces a new binding in the same scope, which behaves like a fresh immutable variable. It does *not* modify the original binding, and both bindings may coexist conceptually during compilation, though only the latest is accessible.
Formally, if `let x = e;` and then `let x = f(x);`, the second `x` is independent of the memory or mutability status of the first.
Therefore, shadowing preserves immutability and supports safe re-binding semantics without aliasing or lifetime issues. ⌟

III Affine Semantics

Definition III.1. Affine Type System.
An *affine type system* is a typing discipline in which each bound variable may be used at most once. Formally, given a context Γ, the judgment $\Gamma \vdash e : \tau$ implies that every variable in Γ is consumed at most once during the evaluation of e. ⌟

Remark 4. (Categorical Affine Semantics of Ownership)
Rust's ownership and borrowing system can be rigorously modeled using typed categorical semantics:

- The type system is *affine*, meaning variables must be used at most once. This disallows contraction but allows weakening.
- Affine logic forbids the structural duplication of assumptions:

$$\text{(Contr)} \quad \frac{\Gamma, x : A, x : A \vdash e : B}{\Gamma, x : A \vdash e : B}$$

  but permits their omission:

$$\text{(Weak)} \quad \frac{\Gamma \vdash e : B}{\Gamma, x : A \vdash e : B}$$

- Categorically, Rust's semantics can be embedded in a *typed symmetric monoidal category* $\mathsf{AffType}$:
  - Objects are types equipped with resource annotations (e.g., $x : T$),
  - Morphisms $f : A \to B$ represent computations consuming A to produce B without duplication,
  - The tensor product $A \otimes B$ models concurrent resource availability.
- Borrowing introduces modal structure:
  - Shared borrows '&T' admit a comonadic modality $\Box$, enabling read-only access,
  - Exclusive borrows '&mut T' model ephemeral, linear use of the resource without aliasing.

⟶

Theorem III.2. No Contraction in Affine Semantics.
Let $\Gamma, x : \tau \vdash e : \sigma$ be a valid typing judgment in an affine type system. Then there does not exist a derivation rule of the form:

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e[x/x_1, x/x_2] : \sigma}{\Gamma, x : \tau \vdash e[x/x_1, x/x_2] : \sigma} \quad \text{(Contraction)}$$

unless the type τ satisfies $\tau \in \mathsf{Copy}$, where Copy is the set of types that admit duplicable semantics under Rust's trait system. ⌟

Proof. In an affine type system, variable bindings are treated as linear resources: each bound variable must be used at most once in the evaluation of an expression. The contraction rule, which allows a variable $x : \tau$ to be duplicated into multiple instances x_1, x_2 within the same typing context, violates this linear usage constraint.

By definition, contraction asserts the admissibility of a structural rule of the form:

$$\text{(Contr)} \quad \Gamma, x : \tau \vdash e[x/x_1, x/x_2] : \sigma \quad \text{where } x_1, x_2 : \tau$$

which would allow two uses of x where only one binding exists.
Such a rule is not derivable in affine systems. In Rust, this semantic restriction is enforced at the type level: only types implementing the `Copy` trait may be duplicated implicitly. For non-`Copy` types, such as `Vec<T>`, `String`, or user-defined types with heap-allocated stat, duplication without explicit cloning constitutes a violation of memory safety and is therefore disallowed.
Thus, in Rust's affine model, the absence of contraction ensures that each instance of a value is used exactly once unless its type is explicitly duplicable. This enforces exclusive ownership and prevents aliasing, double frees, and race conditions. The uniqueness of usage is preserved categorically by modeling each variable-to-expression relationship as a *partial morphism* with strict consumption semantics.
Therefore, contraction is ruled out for general τ, and only recoverable when τ admits duplication under an auxiliary semantic judgment:

$$\tau \in \mathsf{Copy} \quad \Rightarrow \quad \text{Contraction admissible.}$$

□

Problem 3. (Ownership vs. Copy)
In Rust, the following code is valid:

```
let x = 3;          // i32 implements Copy
let y = x;
let z = x;          // x still valid
```

But the following fails:

```
let s1 = String::from("hello");
let s2 = s1;
let s3 = s1;        // error: use of moved value
```

Explain the underlying affine typing reasons. ⌟

Solution. The type `i32` implements the `Copy` trait, so assignment creates a bitwise duplicate without violating affine semantics. In contrast, `String` owns heap-allocated data and does not implement `Copy`; thus, moving it transfers ownership. Reusing `s1` after the move violates the single-use constraint of affine logic.

Proposition III.3. Borrowing as Modal Operation.
Let $\Gamma \vdash e : \tau$ be a well-typed expression in an affine type system. Then forming a shared reference $\&e$ corresponds to a modal operation in the type system:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \&e : \Diamond \tau}$$

where $\Diamond\tau$ denotes a type under a modality that grants permission to observe e without consuming it.
Properties:

1. **Non-consumption:** The original term e remains valid in the context Γ. Borrowing does not move or invalidate ownership.
2. **Duplication:** Values of type $\Diamond\tau$ may be duplicated freely. That is, $\Diamond\tau \vdash \Diamond\tau \otimes \Diamond\tau$.
3. **Read-only Semantics:** No mutation is permitted through $\Diamond\tau$; access is observational only.
4. **Modal Interpretation:** The operator $\Diamond$ behaves like a comonadic modality from linear logic, reflecting duplicability and non-consumption.

Therefore, shared borrowing in Rust aligns with a modal typing rule, where $\Diamond\tau$ encapsulates safe, aliasable observation without affecting the linear flow of ownership. ⌟

Remark 5. (Modal Interpretation)
Borrowing introduces a modality into the type system that distinguishes between consuming and non-consuming access.
Shared borrow: $\&e$ corresponds to a type $\Diamond\tau$, where:

- **Observation is permitted,**
- **Duplication is permitted,**
- **Mutation is prohibited.**

Mutable borrow: $\&\texttt{mut}\, e$ introduces a separate, *affine-exclusive* modality $\blacklozenge\tau$ with the properties:

- **Observation is permitted,**
- **Mutation is permitted,**
- **Duplication is prohibited.**

These modalities reflect a structural refinement of access rights:

$$\textbf{Ownership} \Rightarrow \textbf{Affine Modalities} \Rightarrow \textbf{Usage Permissions}$$

and clarify the design of Rust's borrow checker as enforcing modal constraints over a linear-affine core. ⟶

IV Clifford-Typed Affine Semantics

Rust enforces a strict ownership model via affine types and modal borrowing semantics. We posit that these operational rules reflect deeper geometric and categorical structures; specifically, morphisms in a resource-sensitive category enriched by Clifford-type internal logic.

Conjecture 1. (Omega Typing in Clifford-Typed Affine Semantics)
There exists a unique faithful functor

$$F : \mathsf{Own} \longrightarrow \Omega\text{-}\mathsf{Type}$$

such that each Clifford-typed object admits a grading-preserving, modality-reflecting embedding into an Ω-type, and such that affine and modal constraints in Rust correspond to structural laws in Ω-Type.
Specifically, for every morphism $f : T \to T'$ in Own, where T and T' are Clifford-typed and equipped with affine modality structure, the functor F satisfies:

$$F(f^*(\Diamond T)) = \Diamond F(T), \quad F(f^*(\blacktriangle T)) = \blacktriangle F(T)$$

where $\Diamond, \blacktriangle$ are modal operators internal to the Ω-Type semantic system.
This conjecture implies that the resource-sensitive operations of Rust can be interpreted entirely within the internal logic of a geometric type theory governed by Ω, embedding operational behavior into a coherent, semantic geometric framework. ■

Remark 6. (Coherence Typing as Ω-Type)
This rule will later appear as a coherence law in our Ω-Type model of Rust's ownership system, as part of the categorical semantics framework we seek to build across all possible typed geometric systems—including Clifford-algebraic, cohomological, and modal resource-sensitive logics. ⤚

Let $\Gamma, x : \tau \vdash e : \sigma$. In affine logic, contraction is disallowed unless τ admits duplication (i.e., implements the `Copy` trait).

Theorem IV.1. No Contraction in Affine Semantics.
If $\tau \notin \mathsf{Copy}$, then the rule

$$\text{(Contr)} \quad \Gamma, x : \tau \vdash e[x/x_1, x/x_2] : \sigma$$

is not admissible. ⌟

Proof. This follows from Rust's affine typing discipline: each non-`Copy` value must be used exactly once. Contraction introduces multiple uses of x, violating this constraint.
Formally, in affine type systems, each bound variable may occur at most once in the typing derivation. The contraction rule,

$$\text{(Contr)} \quad \frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \sigma}{\Gamma, x : \tau \vdash e[x/x_1, x/x_2] : \sigma}$$

permits duplication by identifying two occurrences of τ in the context with a single binding x. This rule is admissible only if the type τ supports duplication—namely, if it admits structural copying without invalidating memory semantics.

In Rust, types not marked with the `Copy` trait must be moved or borrowed explicitly. Attempting to apply contraction in such cases results in a violation of the linearity condition: once moved, the original binding becomes inaccessible, and any subsequent use is undefined behavior.
Therefore, the absence of contraction for non-`Copy` types is a direct consequence of enforcing linear usage constraints. The type system prevents the derivation of judgments that would allow multiple evaluations or aliases of a uniquely owned value. □

Let $\Diamond\tau$ denote read-only observational permission.

Proposition IV.2. Borrowing as Modal Operation.
If $\Gamma \vdash e : \tau$, then:

$$\Gamma \vdash \&e : \Diamond\tau$$

where $\Diamond\tau$ denotes a non-consuming, observational modality over τ, preserving linear resource integrity while enabling read-only access. ⌟

Remark 7. (Modal Interpretation)
Borrowing corresponds to a non-consuming modal transition, with $\Diamond\tau$ understood as observational access, while $\blacktriangle\tau$ (if introduced) could denote exclusive mutable access.
In the categorical semantics of resource-sensitive type systems, $\Diamond$ can be interpreted as a comonadic modality: it permits introspection or shared access to a value without duplication or mutation. This aligns with Rust's shared reference model, where an expression of type $\&T$ allows read-only access to memory without violating ownership.
Conversely, $\blacktriangle\tau$ would correspond to a linear or affine monadic modality, encoding exclusive access and single-use constraints. This captures Rust's mutable borrow semantics (`&mut`), where exclusivity guarantees that no aliasing occurs and mutation remains safe.
The interplay between $\Diamond$ and $\blacktriangle$ governs the flow of ownership and the allowable transitions within the typing judgment space. These modal operators enforce invariants across borrow scopes and lifetimes, and their categorical semantics naturally embed into functorial systems tracking resource lineage and state transformation.
This modal layering extends the expressive power of the type system beyond raw syntax, grounding Rust's memory guarantees in formal logical structure and enabling transport of safety properties across semantic transformations. ⟼

Let $\mathcal{C}\ell(V, q)$ be a Clifford algebra over a quadratic space (V, q). We propose to treat such structures as enriched types: allowing compound ownership-sensitive transformations encoded via multivectors and blades.

Definition IV.3. Clifford-Typed Object.
A type $T \in \Omega\text{-}\mathsf{Type}$ is *Clifford-typed* if it is equipped with an internal grading:

$$T = \bigoplus_{k=0}^{\dim V} T^{(k)}$$

and morphisms preserve multivector degree. ⌟

Tracking ownership history amounts to tracking morphisms across time. We propose encoding this lineage

via cohomological chains or simplicial complexes associated to scope and borrow regions.

Definition IV.4. Semantic Cohomology.
Let $\mathcal{U}$ be a cover of lifetime regions. Define a presheaf of borrow permissions $\mathscr{B}$ over $\mathcal{U}$. Then:

$$\check{H}^n(\mathcal{U}, \mathscr{B})$$

classifies *n-fold borrow coherences*.
We define a category Own, whose:

- Objects are typed memory regions $T : \Omega\text{-}\mathsf{Type}$,
- Morphisms are safe ownership-preserving operations (e.g., move, borrow),
- Coherence laws enforce commutativity between modality and mutation.

⌟

Remark 8. (Clifford categories enable geometric typing)
The functorial semantics of $\mathsf{Own} \to \mathsf{Cliff}_k$ (Clifford categories) enables geometric typing, ensuring transformations preserve both logical and memory safety. Each type in Own corresponds to a graded object in the target Clifford category, typically modeled as a multivector space with internal blade decomposition. Morphisms in Own, such as move, borrow, and drop, are mapped to morphisms in Cliff_k that preserve both the algebraic and resource-sensitive structure.
For example, a `move` corresponds to a degree-preserving isomorphism, while an immutable `borrow` corresponds to a non-consuming inclusion morphism into the observational subspace. Mutable borrows may be interpreted as coherent liftings to exclusive morphisms within a tracked subcomplex, preserving blade hierarchy and avoiding aliasing.
Logical safety is guaranteed by ensuring that these morphisms respect the affine typing discipline–i.e., no contraction without duplication semantics. Memory safety arises through coherence laws imposed on compositions: for example, that sequential borrows and returns form contractible paths in the semantic cohomology of lifetime regions.
Thus, geometric typing via Clifford categories extends Rust's operational guarantees into a fully semantic, categorical layer suitable for formal verification, optimization, and eventually, generative type synthesis. ⤚

Remark 9. (Cohomological Structure of $\Omega\text{-}\mathsf{Type}$)
In the $\Omega\text{-}\mathsf{Type}$ framework, each type T is posited as an evolving geometric entity with internal structure, such as grading, modality, and locality, captured categorically. The cover $\mathcal{U}$ of lifetime regions corresponds to an open cover in a topological or site-theoretic sense, where each open set models a local temporal or spatial context in which a borrow operation is valid. The presheaf $\mathscr{B}$ assigns to each region $U \in \mathcal{U}$ the set of borrow permissions active in that context.
The Čech cohomology group $\check{H}^n(\mathcal{U}, \mathscr{B})$ thus encodes higher-order interactions, such as overlapping lifetimes, nested borrows, or reentrant borrow scopes, as cohomological obstructions or coherences. In this sense, an n-cocycle models an n-fold overlapping borrow structure, and a vanishing cohomology class implies global composability of local borrow operations.
This structure integrates with the category Own, where:

- **Objects** $T \in \Omega\text{-Type}$ carry Clifford gradings, modal annotations, and associated local cohomology data.
- **Morphisms** $f : T \to T'$ preserve both ownership semantics and the grading/modality structure, i.e., f respects the decomposition $T = \bigoplus T^{(k)}$, and commutes with modal transitions $\Diamond, \blacktriangle$.
- **Coherence laws** in Own correspond to higher coherence diagrams in $\Omega\text{-Type}$, ensuring that composite borrow flows respect lifetime coverage and do not violate exclusivity.

Therefore, the categorical semantics of ownership emerges as a sheaf-cohomological control structure on top of $\Omega\text{-Type}$, with modal geometry, type safety, and memory correctness enforced by cohomological coherence conditions. $\rightarrowtail$