Thinking Like Rust

Flyxion
Independent Researcher

August 2026

Preface: How to Read This Book

Most books about a programming language teach the language directly: here is a variable, here is a loop, here is how to declare a function. Those books are not wrong to do this, and several of them do it very well. This book tries to do something different, and it is worth being explicit, before Chapter 1 begins, about what that difference actually is and why it might be worth an entire book of its own.

Rust's distinguishing features, ownership, borrowing, lifetimes, exhaustive pattern matching, are usually introduced as a set of rules to be memorized: the compiler enforces them, they prevent certain categories of bugs, and a working programmer eventually internalizes them well enough to stop noticing the friction. That account is true as far as it goes, but it treats the rules as arbitrary constraints that happen to produce good outcomes, the way a strict style guide might. This book's premise is that the rules are not arbitrary. They are the consequences, worked out in detail, of a small number of much older and much more general ideas about what a program is, what state is, why mutation is dangerous, and why correct reasoning about a system should require only a bounded amount of context. Ownership did not have to look the way it looks in Rust; but given the problem it is solving, stated precisely enough, something very like it was close to inevitable. This book tries to earn that word, inevitable, honestly, chapter by chapter, rather than asserting it.

The Vocabulary This Book Uses

A handful of words recur throughout every chapter: *admissibility*, *reachability*, *continuation*, *locality*, *reconstruction*, *boundary*. None of these are Rust-specific terms, and that is deliberate. They are drawn from a broader way of thinking about systems, constraints, and the conditions under which a state transition is legitimate, that this author has developed across a range of other work outside programming. Rust turns out to be an unusually clean, unusually concrete place to apply that vocabulary, because its compiler actually enforces many of the constraints other systems can only aspire to, and because its errors, when something violates an admissibility rule, are unusually precise about naming exactly what went wrong. Readers who have encountered this vocabulary elsewhere will recognize it here; readers encountering it for the first time will find every term defined, formally, at the point it is first needed, and used consistently afterward.

How Each Chapter Is Built

Every chapter in this book follows the same nine-part structure, deliberately, so that once the rhythm becomes familiar it stops requiring attention and the content can take center stage. Each chapter opens with a single motivating question, posed before any Rust appears. It then

works through one or two concrete, non-programming phenomena that exhibit the same underlying structure the chapter is about to formalize, a mathematician's proof, a mechanic's repair, a hospital's sterile field. Only after that grounding does the chapter state a general principle, then a formal definition, then the specific Rust mechanism that realizes it, then worked examples. Every chapter includes one interpretation exercise: a piece of code that compiles, paired with a question asking which admissibility claim it does or does not actually satisfy, since compiling is necessary but never sufficient for correctness in the sense this book cares about. Every chapter closes with a named, one-sentence invariant, stated formally, that the chapter has argued for and that later chapters are free to build on without re-deriving.

The heavy reliance on non-programming analogies throughout, a library book for ownership, a traffic light for enums, a sterile surgical field for `unsafe`, is not decoration. Douglas Hofstadter and Emmanuel Sander have argued at length that analogy-making is not a peripheral cognitive skill deployed occasionally alongside real thought, but close to the core mechanism of thinking itself, the process by which an unfamiliar structure is recognized as an instance of a pattern already understood [7]. This book's chapters are built on that premise directly: a reader who has never written a line of Rust already understands, at some level, why a library only lets one person check out a given book at a time, and Chapter 5's job is not to introduce a new idea from nothing but to help the reader recognize that Rust's ownership rule is the same pattern, already familiar, wearing unfamiliar syntax.

This means the book can be read two ways. Read start to finish, it is a single, cumulative argument: Part I builds the vocabulary from nothing, Part II uses it to derive ownership, Part III extends it to composite types and interfaces, Part IV to partiality and failure, Part V to concurrency, and Part VI to every kind of boundary a Rust program eventually has to cross. Read chapter by chapter as a reference, each chapter's invariant box is a self-contained summary of what that chapter established, and the formal definitions are stated precisely enough to be looked up without re-reading the surrounding prose.

Who This Book Is For

This book assumes the reader is capable of following formal argument and comfortable with basic mathematical notation, sets, functions, the occasional disjoint union, but it does not assume prior Rust experience; every language feature is introduced from first principles, and idiomatic code is shown at every step. It will likely be most rewarding for a reader who already suspects, correctly, that Rust's rules are not arbitrary and wants that suspicion confirmed with actual arguments rather than folklore. It should also be useful to a reader who already knows Rust well and wants a vocabulary for explaining, to themselves or to others, why the language feels the way it does. It is not a syntax reference, and readers who need one are better served by the official documentation; this book's job starts where a syntax reference's job ends.

A Preview, Compressed

It is possible to compress Rust's headline rules into a short list: every value has one owner; you may share it for reading or borrow it for writing, never both at once; there is no null, so a possibly-missing value must be handled explicitly. Summaries along these lines circulate widely, and they are not wrong. They are, however, exactly the kind of claim this book refuses to state without first earning it. "Sharing is either many readers or one writer, which stops data

chaos" is a conclusion with the argument removed; it tells a reader what the rule is without telling them why aliasing combined with mutation, specifically, is the thing that goes wrong, which is the actual content Chapter 3 spends its own effort deriving before Chapter 5 ever states the rule. A reader who only ever learns the compressed version can follow the rule but will be at a loss the first time an unfamiliar compiler error does not obviously match the slogan they memorized; a reader who has followed the derivation will usually be able to work out what the compiler is objecting to even in an unfamiliar case, because they know what the rule is protecting, not just what the rule says. This book's promise is that every rule you might see compressed into a single sentence elsewhere in this preface's spirit will, by the time its chapter ends, have been derived rather than asserted, and a short reference restating the compressed forms, now earned, appears as Appendix B once that derivation is behind you.

Contents

Primer 1: Variables, Mutability, and Basic Types

The chapters that follow use Rust syntax from their very first page, on the assumption that a reader either already knows it or can pick it up as they go. This primer, and the two after it, exist for the reader who wants a quick, conventional pass over that syntax first. Nothing here is derived from admissibility, locality, or any of the vocabulary the rest of the book builds; it is a plain description of what the syntax means, the kind of material a language reference would give you, included so that Chapter 1 can start from a reader who already recognizes `let` and `fn` on sight.

Variables and `let`

A variable is introduced with `let`, and its type is usually inferred from the value assigned to it:

```
let age = 34;           // inferred as i32
let name = "Ada";        // inferred as &str
let pi = 3.14159;        // inferred as f64
```

Variables are immutable by default: once bound, `age` above cannot be reassigned. Writing `age = 35;` on a later line is a compile error. To allow reassignment, the binding must be declared mutable explicitly:

```
let mut age = 34;
age = 35; // fine, because `age` was declared mut
```

This default, immutable unless explicitly marked otherwise, is not a stylistic preference the language happens to have; Chapters 3 and 5 spend considerable effort explaining exactly why it is the correct default, but that argument is not needed to simply read and write the syntax.

Shadowing

Rust also permits shadowing: declaring a new variable with the same name as an earlier one, which creates a distinct binding rather than mutating the original.

```
let x = 5;
let x = x + 1;       // a new binding, also named x, valued 6
let x = x * 2;        // another new binding, valued 12
```

This is different from `mut`: each `let` here introduces a genuinely new variable, and, unlike reassignment, shadowing can even change the type:

```
let spaces = "   ";         // a &str
let spaces = spaces.len(); // now a usize, a different type entirely
```

Basic Scalar Types

Rust's primitive scalar types cover integers, floating-point numbers, booleans, and characters.

```
let a: i32 = -7;         // signed 32-bit integer (the default integer type
    )
let b: u32 = 7;          // unsigned 32-bit integer, cannot be negative
let c: i64 = 9_000_000_000; // signed 64-bit, for larger values
let d: f64 = 2.5;         // 64-bit floating point (the default float type)
let e: bool = true;       // boolean, true or false
let f: char = 'z';        // a single Unicode scalar value
```

Integer types come in signed (`i8`, `i16`, `i32`, `i64`, `i128`) and unsigned (`u8`, `u16`, `u32`, `u64`, `u128`) variants, the number indicating how many bits the value occupies and therefore what range it can represent; `i32` is used unless a program has a specific reason to choose otherwise. Ordinary arithmetic operators, `+`, `-`, `*`, `/`, `%`, work as expected on numeric types, and comparison operators, `==`, `!=`, `<`, `>`, `<=`, `>=`, produce a `bool`.

Comments

A double slash starts a line comment; a slash-star pair, mirrored at the close, starts a block comment.

```
// this line is a comment and is ignored by the compiler
let x = 5; // comments can also follow code on the same line
/* a block comment
   can span multiple lines */
```

A Note on What Comes Next

Chapter 1 will treat a program as a state transition, and by Chapter 5 the word `mut` above will have been given a precise, formal justification rather than simply a description of what it does. This primer's job was only to make the syntax itself unsurprising; the rest of the book explains why it is shaped the way it is.

Primer 2: Control Flow and Functions

`if` as an Expression

An `if` in Rust is an expression, not merely a statement; it produces a value, which is why both of its branches must produce the same type.

```
let number = 6;
if number % 2 == 0 {
    println!("even");
} else {
    println!("odd");
}

let description = if number % 2 == 0 { "even" } else { "odd" };
// description is bound directly to whichever branch ran
```

There is no need for a separate ternary operator, since `if`/`else` already serves that purpose whenever both branches are single expressions.

Loops: `loop`, `while`, and `for`

`loop` repeats a block indefinitely until an explicit `break`; `break` can also return a value from the loop.

```
let mut counter = 0;
let result = loop {
    counter += 1;
    if counter == 10 {
        break counter * 2; // the loop expression evaluates to 20
    }
};
```

`while` repeats as long as a condition holds:

```
let mut n = 3;
while n != 0 {
    println!("{n}");
    n -= 1;
}
```

`for` iterates over anything implementing `IntoIterator`, most commonly a range or a collection:

```
for i in 0..5 {          // 0, 1, 2, 3, 4 -- upper bound exclusive
    println!("{i}");
```

```
}

for i in 0..=5 {          // 0 through 5 inclusive
    println!("{i}");
}

let items = vec![10, 20, 30];
for item in &items {       // borrows each element; items is still usable
    after
    println!("{item}");
}
```

Functions

A function is declared with `fn`, its parameters and their types listed in parentheses, and its return type, if any, given after an arrow.

```
fn add(a: i32, b: i32) -> i32 {
    a + b // no semicolon: this is the function's return value
}

fn greet(name: &str) {
    println!("hello, {name}");
    // no return type declared, so this returns the unit type ()
}
```

The last expression in a function body, if it has no trailing semicolon, is the value the function returns; an explicit `return` keyword is also available and is typically used only for an early return from inside a nested block.

```
fn abs(n: i32) -> i32 {
    if n < 0 {
        return -n; // early return
    }
    n // otherwise, the final expression
}
```

Expressions Versus Statements

Rust distinguishes expressions, which produce a value, from statements, which do not. A block, delimited by `{}`, is itself an expression: its value is the value of its final, semicolon-free line, or the unit type `()` if every line ends in a semicolon or the block is empty.

```
let y = {
    let x = 3;
    x + 1 // no semicolon: this is the block's value
};
// y is 4
```

This is why `if` and `loop` can produce values in the examples above: both are themselves expressions built from blocks, and a block's trailing expression is what the surrounding expression evaluates to.

A Note on What Comes Next

Chapter 1 will treat exactly this distinction, a program as a transition from one state to another, as the book's starting point, and `if`, `match` (introduced formally in Chapter 8), and function calls will all turn out to be different syntactic ways of expressing the same underlying idea. This primer's job was only to make the surface syntax familiar first.

Primer 3: Collections and Strings at a Glance

Tuples

A tuple groups a fixed number of values, possibly of different types, into one compound value; its length is fixed at compile time.

```
let point: (f64, f64) = (3.0, 4.0);
let (x, y) = point;       // destructuring
println!("{} {}", point.0, point.1); // access by position
```

Arrays

An array holds a fixed number of elements of the same type, its length also fixed at compile time and part of its type.

```
let days: [&str; 3] = ["Mon", "Tue", "Wed"];
let zeros = [0; 5]; // an array of five zeros: [0, 0, 0, 0, 0]
let first = days[0];
```

`Vec<T>`: A Growable List

Where an array's length is fixed, `Vec<T>` can grow or shrink at runtime; it is the default choice for an ordered, resizable collection.

```
let mut scores: Vec<i32> = Vec::new();
scores.push(90);
scores.push(85);
scores.push(100);

let scores2 = vec![90, 85, 100]; // shorthand for the same result

for score in &scores {
    println!("{score}");
}

let first = scores[0];        // panics if the index is out of bounds
let maybe_first = scores.get(0); // returns Option<&i32> instead of
   panicking
```

Chapter 13 explains in full why `.get` returning `Option<&i32>`, rather than a bare `&i32` that might not exist, is the more honest of the two designs; both are shown here simply as available syntax.

Strings: `String` and `&str`

Rust has two everyday string types. `&str` is a borrowed, immutable view onto a sequence of UTF-8 text, often a string literal baked into the compiled program; `String` is an owned, growable, heap-allocated string.

```
let greeting: &str = "hello";         // a string slice, borrowed
let mut owned: String = String::from("hello"); // an owned, growable
    String
owned.push_str(", world");            // mutation requires an owned String
owned.push('!');

let combined: String = format!("{greeting}, again");
```

A function parameter typed `&str` can accept both an actual `&str` and a reference to a `String`, since Rust automatically converts the latter where needed; this is why `&str` is the more common choice for a function that only needs to read a string rather than own or grow one.

```
fn print_it(s: &str) {
    println!("{s}");
}

let owned = String::from("hi");
print_it(&owned);   // works: &String converts to &str here
print_it("literal"); // also works directly
```

A Note on What Comes Next

Chapter 5 formalizes exactly why `&str` above is called a *borrow* and why mutating a `String` through `.push_str` requires that nothing else currently hold a reference to it; the vocabulary, ownership, borrowing, lifetimes, is introduced there from first principles rather than assumed. Chapter 6 gives `Vec<T>` and tuples their formal treatment as product state spaces, and Chapter 13 gives `Option`, glimpsed above in `.get`'s return type, a full chapter of its own. This primer's job was only to make the syntax recognizable before any of that argument begins.

Part I

Thinking Like Rust

Chapter 1: What Is a Program?

From First Principles

Question: Is a program the text you type, or the behavior that text produces?

1.1 Phenomenon: The Same Behavior, Different Text

Two recipes can produce the same dish through different sequences of steps: one cook creams the butter and sugar before adding eggs, another beats the eggs first and folds them in afterward. A diner who tastes the finished cake cannot recover, from the cake alone, which order the cook used. What survives, and what actually mattered to the diner, was not the sequence of motions in the kitchen but the transformation from raw ingredients to finished dish.

The same separation appears in arithmetic. A student computing $2 \times (3 + 4)$ might first add and then multiply, or might distribute the multiplication across the sum and then add the products. Both routes end at 14. The routes differ; the function from inputs to output does not. Anyone who only cares about the result, and not about which route was taken to it, is implicitly treating the calculation as a function rather than as a particular sequence of steps.

Two programs can behave identically while looking nothing alike on the page. A function that sums a list with a loop and a function that sums the same list with recursion produce the same output for every input, despite sharing almost no textual structure. If a program were identical to its text, these would have to count as two different things. Most working programmers' intuitions disagree: they would call these the same computation, expressed two ways. That intuition is worth taking seriously, because it is quietly making a claim about what a program actually is.

The same pattern appears well outside software. A thermostat might be built around any number of different control algorithms, a simple on-off threshold, a proportional controller, a more elaborate predictive scheme, and yet every one of them realizes the same transition: given a room's current temperature, bring it toward a target range. A calculator user who presses one sequence of buttons and a user who presses an entirely different sequence can both arrive at the same displayed result, and the calculator does not consider the second user to have used a different machine. A train journey planned along one route and the same journey planned along a different route, differing in every intermediate station, still realize the same transition, if both begin at the same origin and end at the same destination. A deck of cards sorted by insertion sort and the same deck sorted by merge sort pass through entirely different intermediate arrangements, and finish in exactly the same order. In each case, the intermediate procedure is free to vary; what stays fixed, and what actually mattered to whoever cared about

the outcome, is the transition from starting condition to ending condition.

1.2 General Principle: A Program Is a Transition, Not a Transcript

The examples above share a structure. In each, there is a starting condition, an ending condition, and a claim that many different intermediate routes can connect the same starting condition to the same ending condition without the identity of the underlying transformation changing. Call the starting condition and ending condition together a *state transition*, and call the intermediate route the *implementation* of that transition. The recipe, the arithmetic method, and the summing function are all implementations; the transformation they realize, from raw ingredients to cake, from operands to product, from an unsummed list to a total, is the thing that has an identity independent of any one implementation.

This suggests a working answer to the chapter's opening question. A program is not most usefully identified with its text, the particular sequence of instructions a compiler happens to read, because two different texts can realize the identical transformation, and a definition of "program" that could not see this would be drawing its boundaries in the wrong place. A program is better identified with the state transition it realizes: a mapping from whatever condition holds before it runs to whatever condition holds after. The text is one implementation of that mapping, useful and necessary, but not the thing itself.

1.3 Formal Definition: State and Transition

Let Σ denote the set of all states a system could be in, everything that could vary: the contents of memory, the values of variables, the position of a file cursor, whatever is relevant to the system under discussion. A *program* P, in the sense this book will use throughout, is a (possibly partial) function

$$P : \Sigma \to \Sigma$$

mapping an initial state $\sigma \in \Sigma$ to a resulting state $P(\sigma)$. Two pieces of program text, T_1 and T_2, are said to be *observationally equivalent* if they realize the same function P, that is, if $T_1(\sigma) = T_2(\sigma)$ for every σ on which both are defined, regardless of how different T_1 and T_2 look, and regardless of how many intermediate steps each takes to get there.

This definition is intentionally silent about performance, memory usage, and the number of steps taken. Those are real properties of an implementation, and later chapters, particularly the ones on performance and locality, take them seriously. But they are properties of *how* a transition is realized, not properties of *which* transition is realized, and the chapter's argument depends on keeping that distinction available before folding performance concerns back in.

1.4 Rust Mechanism: Functions as Named Transitions

Rust's function syntax makes the state-transition view unusually explicit compared to many languages, because a function signature is required to declare, up front, exactly what it consumes and exactly what it produces. A function such as `fn sum(v: Vec<i32>) -> i32` is a direct, readable statement of a transition: from a state containing a `Vec<i32>`, produce a state containing an `i32`, with the vector itself consumed in the process. The signature is not

merely documentation; later chapters will show that the compiler treats it as a binding contract, checked at every call site.

```
fn sum_loop(v: &Vec<i32>) -> i32 {
    let mut total = 0;
    for x in v {
        total += x;
    }
    total
}

fn sum_fold(v: &Vec<i32>) -> i32 {
    v.iter().fold(0, |acc, x| acc + x)
}
```

1.5 Worked Example: Two Implementations, One Transition

`sum_loop` and `sum_fold` above are textually unrelated: one uses explicit mutation and a `for` loop, the other uses an iterator combinator with no mutable variable in sight. Despite this, both realize the same function from `&Vec<i32>` to `i32`: for every vector, both produce the same total. In the vocabulary of Section 1.3, `sum_loop` and `sum_fold` are observationally equivalent implementations of a single underlying transition. Neither is more entitled to be called "the" sum function than the other; the transition is the invariant, and the two functions are two witnesses to it.

1.6 Interpretation Exercise: When Two Implementations Are Not Equivalent

```
fn sum_with_log(v: &Vec<i32>) -> i32 {
    let mut total = 0;
    for x in v {
        println!("adding {x}");
        total += x;
    }
    total
}
```

This function returns the same `i32` as `sum_loop` for every input. Is it observationally equivalent to `sum_loop` in the sense defined above? The question worth sitting with is: *what counts as part of the state* Σ*?* If Σ is taken to include only the function's argument and return value, the two are equivalent, since both compute the same total. If Σ is taken to include the program's standard output stream, they are not, since `sum_with_log` leaves a trace that `sum_loop` does not. Section 1.3's definition did not specify what belongs in Σ because there is no single correct answer; choosing what counts as state is itself a modeling decision, and later chapters, especially the ones on interfaces and boundaries, return repeatedly to the consequences of drawing that boundary one way rather than another.

Recurring Example: The Library System

A thread will run through the remainder of Part I and into Part II: a small library's book-checkout system, returned to at the close of each chapter to show that chapter's idea applied to one continuous, growing example rather than a fresh, unrelated one each time. Here, at the start, the example is deliberately minimal. Checking a book out is a state transition in exactly Section 1.3's sense: a function from a state in which the book is available to a state in which it is checked out to a particular borrower, $P : \Sigma \to \Sigma$. Whether the librarian scans a barcode, types the title into a terminal, or writes it on a paper card, the transition realized is the same one; the card and the barcode scanner are two implementations of an identical transformation, exactly as Section 1.5's `sum_loop` and `sum_fold` were two implementations of the same sum. Later chapters will build this example up considerably; for now, the point is only that "a book gets checked out" is best understood as a transition between two states of a record, not as a sequence of physical or digital actions that happens to produce that transition.

Connection to Category Theory

A program's identity as a transition between states, argued for at length in this chapter, is what category theory calls a *morphism*: an arrow from one object to another, carrying no essential content beyond the objects it connects. Readers curious to see this abstraction developed on its own terms, rather than in service of a systems argument, may consult Bartosz Milewski's *Category Theory for Programmers* [9], which begins from exactly this notion and builds outward. This book deliberately postpones that vocabulary until each underlying idea has already been derived from concrete cases; Appendix F collects the full correspondence once the derivations are complete.

1.7 Connections: Why This Framing Matters Before Rust-Specific Content

Treating a program as a state transition, rather than as a transcript of instructions, is what allows the rest of this book to ask questions like "what states can legitimately follow this one" without first committing to a particular syntax. Ownership, covered starting in Chapter 5, is a claim about which parts of Σ a given piece of code is authorized to transform. Traits, covered in Part III, are claims about which transitions a type promises to support without revealing how. Concurrency, covered in Part V, is what happens when more than one transition is in flight over the same Σ at once. None of these later chapters would have stable footing if "program" still meant, informally, "a sequence of lines that run in order"; the transition-based framing established here is the foundation the rest of the book builds on.

Invariant Established in This Chapter

Computation is structured state transformation. A program's identity lies in the mapping it realizes from initial state to resulting state, not in the particular sequence of steps, or the particular syntax, used to realize that mapping. Two implementations that realize the same mapping are, for the purposes of this book, the same program, and the question of which parts of the world belong to that mapping's state is a modeling choice made explicit rather than left implicit.

Chapter 2: Programs as State Transformations

From First Principles

Question: If two programs always agree on their output, is there ever a good reason to prefer one over the other?

2.1 Phenomenon: Composing Without Re-Deriving

An assembly line does not re-derive the design of each part it installs. A worker bolting on a wheel treats the wheel's supplier as a black box: given a wheel meeting the specification, the worker's job is well-defined regardless of how that wheel was forged, machined, or painted. The line as a whole is built by composing many such black-box stages, each trusting the stage before it to have honored its own specification.

A postal system does not re-derive the contents of a letter to decide how to route it. It reads an address, a state visible on the envelope, and forwards the letter accordingly. What happens inside the envelope is irrelevant to the routing decision; two letters with identical addresses and different contents are routed identically. The system composes correctly precisely because routing depends only on the declared, externally visible state, the address, and not on the letter's internal history.

Multi-stage pipelines exhibit the same pattern across a wide range of domains. A water treatment plant moves water through filtration, then disinfection, each stage trusting that the water arriving from the previous stage actually meets that stage's declared input specification, rather than re-testing the water's entire history at every step. A manufacturing line converting ore into steel and steel into a finished car composes in exactly the same way, each stage's process engineers designing to a specification, not to the specific furnace or mill that happened to produce the input. An image-processing pipeline that loads, resizes, and compresses an image treats each stage as a black box characterized only by its declared transformation. A hospital's admission-to-discharge workflow, patient admitted, diagnosed, treated, discharged, composes reliably only to the extent that each stage trusts the previous stage's declared output, a diagnosis, a treatment plan, rather than independently re-verifying everything that happened earlier in the patient's stay.

Software composition works the same way whenever it works well. A caller of a function does not re-derive the function's internals to decide whether to call it; the caller trusts the function's declared behavior, its transition from input state to output state, and builds on top

of that declaration. Chapter 1 established that a program is best identified with the transition it realizes rather than with its text. This chapter asks what follows from taking that identification seriously when programs are combined into larger programs, the way wheels are bolted onto assembly lines and letters are routed by address.

2.2 General Principle: Composition of Transitions

If a program P_1 realizes a transition from states of type A to states of type B, and a program P_2 realizes a transition from states of type B to states of type C, then running P_1 followed by P_2 realizes a transition from A directly to C. This is unremarkable as a statement about functions, $P_2 \circ P_1$, but it is worth stating explicitly because it is the reason large programs can be built at all: a programmer composing P_1 and P_2 needs to know only their declared input and output types, not their internal steps, in order to predict the behavior of the composition.

This is only true, however, if P_1's output actually determines everything relevant to P_2's behavior. If P_2's behavior depends on some part of the state that P_1 did not account for in its declared output type, on some hidden global variable, some file left open, some clock reading captured at an unpredictable moment, then composing P_1 and P_2 by type signature alone silently stops being reliable, and the assembly-line and postal-routing analogies break down: the worker would need to inspect the wheel's manufacturing history after all, and the postal system would need to open the envelope. Section 2.1's examples work because the specification each stage declares is also the specification each later stage actually depends on. Where that alignment fails, composition by declared type becomes an illusion.

2.3 Formal Definition: Referential Transparency

A program P is *referentially transparent* with respect to a state space Σ if $P(\sigma)$ depends only on σ, and nothing outside σ, for every $\sigma \in \Sigma$. Given the composition $P_2 \circ P_1$, the whole composition is predictable from the two declared transitions alone exactly when both P_1 and P_2 are referentially transparent with respect to a shared, correctly-scoped Σ. If either program's actual behavior depends on some state outside the declared Σ, unowned mutable state reachable from elsewhere, an ambient global, an implicit side channel, then the composition's behavior can no longer be predicted from the two signatures alone, and the promise made in Section 2.2 is broken.

This gives a precise sense in which Chapter 1's claim, that a program is its transition rather than its text, has teeth: the claim is only trustworthy to the extent that the transition's declared domain and codomain actually capture everything the implementation depends on. A program whose real behavior silently exceeds its declared Σ is one whose transition identity, in the sense Chapter 1 introduced, has quietly become larger than what its signature claims.

2.4 Rust Mechanism: Signatures as Declared State

Rust's function signatures are unusually strict about naming exactly which parts of Σ a function may depend on and produce, because every value a function touches must appear, by ownership or by reference, somewhere in its parameter list or its surrounding closure capture. A function that wanted to depend on some ambient variable not passed in, the way a

global in many other languages silently can, has to go out of its way to do so, typically through an explicit `static` item, which later chapters will show is treated by the compiler as visibly exceptional rather than the default case.

```
fn apply_discount(price: f64, rate: f64) -> f64 {
    price * (1.0 - rate)
    // Everything this function depends on is named in its
    // signature. Nothing about the surrounding program, no matter
    // how large it grows, can change this function's behavior
    // for a given (price, rate) pair.
}
```

2.5 Worked Example: Composing Two Referentially Transparent Stages

```
fn apply_discount(price: f64, rate: f64) -> f64 {
    price * (1.0 - rate)
}

fn apply_tax(price: f64, tax_rate: f64) -> f64 {
    price * (1.0 + tax_rate)
}

fn checkout_total(price: f64, discount: f64, tax: f64) -> f64 {
    apply_tax(apply_discount(price, discount), tax)
}
```

`checkout_total` can be understood entirely from the declared signatures of `apply_discount` and `apply_tax`, without reading either function's body. This is Section 2.2's composition principle exercised directly: the reader composes two trusted transitions, $A \to B$ and $B \to C$, into $A \to C$, and the reasoning required is proportional to the two signatures, not to the two implementations.

2.6 Interpretation Exercise: A Composition That Looks Safe but Isn't

Before the code, it is worth seeing this failure in a setting with no compiler at all. Suppose you hand a friend a grocery list and a twenty-dollar bill, and ask them to buy what is on the list. If they come back with the groceries, having spent only the twenty, the transaction was referentially transparent in exactly Section 2.3's sense: the declared inputs, the list and the twenty dollars, were the whole truth about what the errand depended on. Now suppose you had also left your credit card sitting on the kitchen counter, and, faced with an item that cost more than twenty dollars, your friend quietly used it to cover the difference. From your side of the front door, the outcome looks identical: they hand you the groceries, the errand appears to have succeeded exactly as declared. But the errand's real footprint was larger than what you gave it, and the difference only surfaces later, on a statement you were not shown at the time. This is the shape of the danger a hidden global dependency creates in code: two calls that look, from their signatures, like the same declared transaction can produce silently different real-world costs, and the discrepancy is invisible at exactly the moment it would have been cheapest to catch.

```
static mut TAX_RATE: f64 = 0.08;

fn apply_discount(price: f64, rate: f64) -> f64 {
    price * (1.0 - rate)
}

unsafe fn apply_tax(price: f64) -> f64 {
    price * (1.0 + TAX_RATE)
}
```

`apply_tax` here no longer declares its dependence on a tax rate in its signature; it reaches out to `TAX_RATE` instead. The question worth asking is not whether this compiles, since with `unsafe` it does, but: *which composition guarantee from Section 2.2 has silently failed?* A caller composing `apply_discount` and `apply_tax` by their signatures alone would have no way to know that the second function's result depends on mutable state elsewhere in the program; two calls with identical arguments could now return different results if `TAX_RATE` changed between them. The composition is no longer predictable from the declared types, exactly the failure mode Section 2.3 defined referential transparency to rule out, and Rust's `unsafe` marker is, once again, the visible admission that this particular function's real Σ has silently grown larger than its signature.

A Further Worked Example: Closures and Iterator Chains as Composition

Two everyday Rust idioms are worth reading through this chapter's vocabulary directly, since both are, at bottom, applications of Section 2.2's composition principle rather than separate topics. A closure is a function value that captures part of its surrounding environment, and Rust's three closure traits, `Fn`, `FnMut`, and `FnOnce`, are themselves a declaration of exactly what the closure's body actually does with what it captured: `Fn` declares read-only access to the captured state, `FnMut` declares a need to mutate it, and `FnOnce` declares that the closure consumes it outright and can only be called once. A function accepting a closure bounded by `Fn` is, in Section 2.3's vocabulary, declaring its own honest dependency on the closure's capture behavior, precisely the same discipline Section 2.4 described for an ordinary function signature, now extended to a value that happens to carry code along with its state.

```
fn apply_all<F: Fn(f64) -> f64>(prices: &[f64], f: F) -> Vec<f64> {
    prices.iter().map(|&p| f(p)).collect()
}

let discounted = apply_all(&[100.0, 50.0], |p| p * 0.9);
```

An iterator adapter chain, `v.iter().map(...).filter(...).sum()`, is composition in exactly Section 2.2's sense, made visible in the syntax itself: each adapter is a small transition, from one iterator state to another, and the chain as a whole composes those transitions the same way `checkout_total` composed `apply_discount` and `apply_tax` in Section 2.5, except that here the intermediate stages are themselves ordinary values a reader can name, inspect, and reorder. Chapters 9 and 10 will formalize traits and generic bounds in full; what is worth noticing here, before either chapter arrives, is that an adapter chain's reliability rests on the same principle this chapter has already established: each stage's declared input and output type is the complete truth about what it depends on and produces, and nothing about the chain's correctness requires reading any adapter's internal implementation.

Historical Example: Apple's "goto fail" Bug

Chapter 8 examines Apple's 2014 TLS verification bug as a failure of pattern-matching discipline; the same bug is worth a second look here, from the composition side, because it is exactly as much a failure of this chapter's subject as of Chapter 8's. A secure connection's establishment is itself a composed state transition, in Section 2.2's sense:

$$\text{Untrusted Certificate} \rightarrow \text{Hash Verification} \rightarrow \text{Signature Verification} \rightarrow \text{Trusted Connection}.$$

Each stage was meant to compose the way `checkout_total` composed `apply_discount` and `apply_tax` in Section 2.5: the connection could only reach "trusted" by actually passing through every intermediate stage, each stage trusting that the one before it had genuinely run to completion. A single duplicated line, an unconditional jump sitting where a conditional one belonged, caused the signature-verification stage to be skipped entirely on every execution, not merely to fail silently but to never run at all. The composed transition Untrusted $\rightarrow$ Trusted no longer required passing through every intermediate stage the protocol's design demanded; it required only the stages that happened to still execute before the accidental jump.

This is worth stating plainly, because it is easy to file the bug under cryptography and move on: the bug was not a failure of any cryptographic algorithm. Every hash and signature routine involved worked exactly as specified. The failure was that the composition connecting them, the sequence in which they were supposed to run, could be silently truncated by a single misplaced control-flow statement, and nothing in the surrounding code's structure made that truncation visible. Section 2.2's composition principle is not a convenience for readability; a caller relying on a composed sequence of stages is relying on the sequence actually running to completion, and a language whose control-flow constructs make silent truncation easy to introduce by accident is a language where that reliance is harder to trust than it looks.

Recurring Example: The Library System

Checking a book out and later returning it are two transitions that compose, in exactly Section 2.2's sense: `checkout` carries a book from *available* to *checked out to a borrower*, and `return` carries it back. A librarian relying on this composition, that a returned book is once again available, is trusting that `return` depends only on the book's own record, not on some other, unstated piece of state, the identity of whoever happens to be at the front desk, say, or which terminal the action was entered on. If `return` secretly consulted something beyond the book's declared record, the composition `checkout` then `return` could fail to restore *available* in a way neither function's signature would reveal, exactly Section 2.3's referential-transparency requirement stated in library terms. The workflow only composes reliably because each stage's declared dependency, the book's own record, is also its complete, honest dependency.

Connection to Category Theory

Section 2.2's composition principle, that two transitions compose into a single transition whose behavior is fully determined by the two it combines, is category theory's composition operation, and any transition that changes nothing, mentioned only in passing so far, is exactly a category's identity morphism. A category, in the formal sense, consists of little more than objects, morphisms between them, and these two properties, composition and identity, together with the requirement that composition be associative [9]. Appendix F extends this correspondence across the rest of the book.

2.7 Connections: Why This Matters for Everything After

Every later chapter in this book relies on the composition principle established here. Ownership and borrowing, starting in Chapter 5, are mechanisms for keeping a function's real dependencies aligned with its declared signature even in the presence of shared, mutable data, which is precisely the situation where alignment is hardest to maintain by convention alone. Traits, in Part III, extend this same alignment to behavior that varies by type. Concurrency, in Part V, is the composition principle applied to transitions that may be interleaved rather than sequenced. The recurring question across all of these, does this function's declared state actually capture everything it depends on, is the same question this chapter posed about `apply_tax`, asked again at larger and larger scales.

Invariant Established in This Chapter

A program's behavior should be fully predictable from the transition it declares, its input and output state, without requiring inspection of its implementation or of anything left unnamed in its signature. Composition of programs is trustworthy exactly to the extent that each component is referentially transparent with respect to a correctly-scoped state space; wherever real dependencies silently exceed declared ones, composition by signature alone stops being reliable, and later chapters will show ownership and borrowing as Rust's primary mechanism for keeping the two aligned.

Chapter 3: Mutation, Constraints, and the Cost of Global Knowledge

From First Principles

Question: If nothing ever changed, would there be any need for rules about who is allowed to change it?

3.1 Phenomenon: Trust That Erodes After the Fact

A library book that only ever gets read imposes almost no coordination burden on the library; any number of people can read the same edition's contents without interfering with one another, so long as none of them is also allowed to rewrite the pages. The moment a library permits patrons to annotate books in ink, the calculus changes completely: every future reader now needs to know whether the copy in their hands has been altered, by whom, and whether those alterations can be trusted, before they can trust anything they read in it. The mutation did not just change the book; it changed what every subsequent reader is now obligated to check.

A shared calendar behaves the same way. A calendar that only ever gets viewed requires no coordination among its viewers. The moment it becomes editable by multiple people, every viewer's prior belief about what a given time slot holds becomes provisional, valid only until the next edit, and stale beliefs about the calendar become a live source of double-booked rooms and missed meetings. The problem was never viewing; it was that an edit made by one party silently invalidates an assumption another party had already formed and was still relying on.

The same pattern shows up anywhere one party's mutation runs ahead of another party's belief. A collaboratively edited document changing while someone else is reading it can leave the reader working from a passage that has since been rewritten out from under them. Rearranging books on a shelf while someone else is navigating from memory turns their prior mental map into a source of wasted searching, or a wrong conclusion, rather than a shortcut. A chess board altered while an opponent is planning several moves ahead invalidates every line of analysis built on the board's earlier position. A restaurant menu changing mid-service, an item running out, a price updating, leaves any order already placed against the old menu resting on an assumption the kitchen can no longer honor. None of these mutations are wrong to make; the problem in each case is only ever the gap between when the mutation happens and when the party relying on the old state finds out about it.

Chapter 1 established that a program is a state transition; Chapter 2 established that com-

posing programs reliably requires that each one's declared state actually capture what it depends on. This chapter asks the question those two chapters were quietly setting up: what happens to a state σ once something is permitted to change it out from under an observer who has already formed a belief about it, and what would it cost to make that safe in general.

3.2 General Principle: Mutation as an Assumption-Invalidating Act

Call an act of *observation* anything that forms a belief about a state's current contents without altering them, and call an act of *mutation* anything that changes the state's contents. An observation, by itself, never invalidates another party's beliefs, because it changes nothing; any number of parties can observe the same unchanging state and remain in agreement indefinitely. A mutation is categorically different: it can invalidate every belief, held by every other party, that was formed by observing the state *before* the mutation occurred, whether or not those parties are aware the mutation happened.

This asymmetry is the entire reason mutation needs rules and observation, largely, does not. A system that permits unrestricted mutation without informing every party who might be relying on the old state is a system that has silently taken on an obligation it has not discharged: the obligation to ensure no one is still depending on the state that just changed. Discharging that obligation requires either preventing the mutation until it is safe, the strategy Rust's ownership system takes at compile time, or checking at the moment of use whether the state one is relying on is still the state one thinks it is, the strategy a version-controlled document or a database transaction takes at runtime.

3.3 Formal Definition: Reachability After Mutation

Recall from Chapter 4's forthcoming vocabulary, introduced informally here and formalized fully in Chapter 5, that $R(v)$ denotes the set of program paths capable of reaching a value v. Say a mutation of v at some point t is *safe with respect to an observer* $p \in R(v)$ if p's beliefs about v, formed by observations prior to t, are not relied upon by p at any point after t without p first re-observing v. A mutation is *globally safe* if it is safe with respect to every $p \in R(v)$, and *globally unsafe* otherwise.

The cost of guaranteeing global safety for a given mutation scales with $|R(v)|$: the more paths that can reach v, the more observers whose beliefs must be accounted for before the mutation can be certified safe. This is the precise sense in which mutation, unlike observation, imposes a cost that grows with the size of the surrounding system rather than staying fixed, and it is the direct link between this chapter and the next: Chapter 4 will name this cost explicitly as a failure of locality, and will ask what it would take to bound it rather than pay it in full every time.

3.4 Rust Mechanism: Refusing the Mutation Rather Than Auditing It

Rust's response to Section 3.3's cost is not to make mutation cheaper to audit; it is to shrink $R(v)$ at the moment of mutation so that the audit becomes trivial. The aliasing rule, previewed

here and developed fully in Chapter 5, guarantees that whenever a mutation through `&mut T` is legal, no other path in $R(v)$ can be simultaneously holding a belief about v's prior state, because the compiler will not allow a shared reference to coexist with a mutable one. The safety obligation from Section 3.2 is discharged not by checking every observer, but by guaranteeing there are no other observers to check.

```
fn overwrite(v: &mut Vec<i32>) {
    v.clear();
    v.push(99);
    // At this point in the program, no other path can hold a
    // belief about v's prior contents: the compiler has already
    // ruled out any coexisting shared reference. The audit
    // Section 3.3 describes is unnecessary because its premise,
    // an observer relying on stale state, cannot arise here.
}
```

3.5 Worked Example: The Cost Made Visible Without the Guarantee

```
fn describe_then_clear(v: &Vec<i32>, cleared: &mut Vec<i32>) {
    let summary = format!("{} items", v.len()); // a belief about v
    cleared.clear(); // mutates a different vector, not v
    println!("{summary}"); // v's summary is still trustworthy
}
```

Here two separate vectors are involved, and the mutation is directed at `cleared` while the belief, `summary`, was formed by observing `v`. Because the two are distinct values with disjoint reachability sets, the mutation of `cleared` cannot invalidate any belief formed about `v`, and Section 3.2's safety condition holds trivially. The example is worth noticing precisely because it shows the compiler is not restricting mutation in general, only mutation that could actually reach a value some other path still depends on.

3.6 Interpretation Exercise: Where the Compiler Draws the Line

```
let mut v = vec![1, 2, 3];
let summary = format!("{} items", v.len());
v.push(4);
println!("{summary}");
```

This compiles, even though it observes `v`, then mutates it, then uses a value derived from the earlier observation. The question worth asking is: *why does this not violate Section 3.2's safety condition, when the earlier shared-reference example in Chapter 5 does?* The answer is that `summary` is a new, independent `String`, not a reference into `v`; by the time `v.push(4)` runs, nothing in `summary` still reaches `v` at all, so $v \notin$ the set of things `summary` depends on. The mutation is safe not because the compiler tracked and cleared some observer, but because `format!` already discharged the dependency by copying the needed information out of `v` before the mutation ever occurred. The distinction between still depending on a value and having already extracted what one needed from it is exactly the distinction Chapters 5 and 6 will formalize as the difference between borrowing and owning.

Recurring Example: The Library System

Two librarians, at different terminals, both pull up the record for the same book at nearly the same moment. Both records show it as available; both librarians proceed to check it out, one to a patron standing at the desk, the other to a hold request submitted online seconds earlier. Whichever write reaches the record second overwrites the first, and the library now believes the book has one borrower while its physical copy has left the building with another. Neither librarian did anything individually wrong; each acted correctly on the belief the record showed at the moment they read it. The failure is exactly Section 3.2's asymmetry: the first librarian's read formed a belief, "available," that the second librarian's write silently invalidated, and nothing in the system's design ensured the second librarian's action accounted for a belief that had already gone stale by the time it was acted on.

3.7 Connections: Setting Up Locality and Ownership

This chapter has deliberately stopped short of describing Rust's actual borrowing rules in full; that is the next two chapters' work. What this chapter has established is the cost that those rules exist to pay down: mutation, left unconstrained, forces every observer of a value to either be tracked and notified or to risk relying on stale beliefs, and the cost of tracking scales with how widely the value is reachable. Chapter 4 names the general principle that would make this cost bounded rather than unbounded, and Chapter 5 shows the specific mechanism, unique authority over mutation, that Rust uses to realize it.

Invariant Established in This Chapter

Mutation changes future reachability. An act of observation leaves every other observer's beliefs intact, but an act of mutation can invalidate the beliefs of every party that reached the value beforehand, whether or not those parties are aware of it. The cost of guaranteeing a mutation is safe scales with the number of paths that can reach the mutated value, and a language must either pay this cost by auditing every observer at the moment of mutation or avoid it by guaranteeing, structurally, that no such observer can exist when the mutation occurs.

Chapter 4: Why Locality Is Valuable

From First Principles

Question: Why should understanding whether a piece of code is correct ever require reading the rest of the program?

4.1 Phenomenon: Reasoning That Does Not Scale

A mathematician proving a theorem rarely reasons about the entire edifice of mathematics at once. A proof is built from lemmas, and each lemma is verified using only the finitely many prior results it explicitly cites. Once a lemma is established, it can be used again without re-deriving it, and, crucially, without re-checking the argument that established it. A proof that could only be verified by simultaneously holding every previously proven theorem in mind would not be a usable proof at all; it would be a single, monolithic argument masquerading as a structured one.

A mechanic replacing a worn belt does not need to disassemble the engine to confirm the repair is correct. The belt's specification, its length, its tension, the diameter of the pulleys it wraps, is enough to determine whether the replacement is admissible. The rest of the engine is relevant to the car's overall operation, but it is not relevant to whether *this particular repair* was done correctly. A repair procedure that required inspecting every other component before any single component could be trusted would make cars unmaintainable in practice, however sound it might be in principle.

A city can often be understood neighborhood by neighborhood. Knowing that a given block is zoned residential, has reliable water pressure, and sits outside the floodplain lets a resident reason about that block without first modeling the entire municipal water and drainage system. The neighborhood's local properties are not merely correlated with the city-wide system; in the cases that matter for the resident's decision, they are *sufficient*, and the rest of the city's structure can be treated as a black box.

The same bounded-reasoning pattern is what makes several other everyday trades and designs tractable. An electrician diagnosing a tripped circuit breaker works from that one breaker and the circuit it protects, not from a mental model of the entire city's power grid. A plumber repairing a single faulty valve does not need to trace every pipe in the building before trusting that the repair is correct. Standardized LEGO studs let two bricks connect correctly based only on the interface the studs define, with no need to know anything about how either brick's interior was molded. A well-drafted legal contract is written so that each clause can be interpreted on its own terms, without requiring a court to hold every other clause in the same

document in mind simultaneously just to determine what one clause obligates.

It is worth imagining, briefly, what any of these trades would look like without this property, since the absence is where locality's value becomes hardest to miss. Picture a city whose plumbing had no locality at all: no valve's behavior could be trusted in isolation, because a pressure spike from a toilet flushing five miles away might, for reasons buried somewhere in the city's entire pipe network, affect the fitting directly under your own kitchen sink. Fixing that fitting would require, in principle, accounting for every other tap in the city before trusting the repair, a task no plumber, and no team of plumbers, could complete in a human lifetime. This is precisely the failure software engineers call *action at a distance*: a property that can only be verified globally, by inspecting an unbounded and constantly changing rest-of-the-system, rather than locally, from a bounded neighborhood the way Section 4.2 is about to formalize. A system exhibiting it at scale is not merely inconvenient to maintain; it is, for practical purposes, unmaintainable, since the cost of verifying any single change grows with the size of the whole system rather than staying fixed.

In each case, the pattern is the same. Some property of a part of a larger system, a lemma, a repair, a block, a circuit, a valve, a brick, a clause, can be verified using a bounded amount of information about that part, without requiring an unbounded inspection of the whole. This is not a convenience; it is what makes reasoning about large systems possible at all. A software function is no different from a lemma, a repair, or a city block in this respect, and the question this chapter is building toward is what, specifically, has to be true of a function for it to admit this kind of bounded reasoning.

4.2 General Principle: Bounded Context

Call the total state of a system Σ. A property P of some component c of that system is *locally verifiable* if there exists some bounded neighborhood $N(c) \subseteq \Sigma$, one whose size does not grow with the size of Σ itself, such that verifying P requires only $N(c)$, and is guaranteed to hold regardless of what the rest of $\Sigma \setminus N(c)$ contains.

Contrast this with a property that is only *globally verifiable*: one whose truth cannot be determined without inspecting all of Σ, because some part of $\Sigma \setminus N(c)$ could, in principle, invalidate it. The mathematician's lemma, the mechanic's repair, and the resident's block are all cases where a locally verifiable property was available and was used instead of a globally verifiable one. The distinction matters because the cost of verification scales very differently in the two cases: local verification costs something proportional to $|N(c)|$, a constant with respect to the size of the whole system, while global verification costs something that grows with $|\Sigma|$, and in the worst case must be redone every time any part of Σ changes.

Software has a specific, well-known name for the failure mode that arises when properties that should be locally verifiable turn out to be only globally verifiable: it is usually called *action at a distance*, and it is the reason large codebases without strong locality guarantees become progressively harder to modify as they grow, even when no individual change is large. Every function added to such a codebase is a new part of $\Sigma \setminus N(c)$ for every existing function c, and if the language provides no mechanism to bound $N(c)$, every function's correctness becomes, silently, a global property of the entire program.

4.3 Formal Definition: Locality of a Function

For a function f operating on some value v, define f's *reasoning footprint* as the bounded neighborhood

$$N(f, v) \subseteq \Sigma$$

that must be inspected to determine whether f behaves correctly with respect to v. A language exhibits *locality* for a class of functions if $N(f, v)$ can always be constructed from f's own signature and body alone, independent of $|\Sigma|$ and independent of which other functions happen to exist elsewhere in the program. A language lacks locality for that class if $N(f, v)$ can only be constructed by additionally inspecting some unbounded portion of $\Sigma \setminus \{f\}$, such as every other function that might, at runtime, hold a reference to v.

This definition is deliberately phrased so that it does not yet mention Rust, ownership, or any particular mechanism. It states a property a language either has or lacks. The remainder of this chapter is an argument that Rust's ownership and borrowing rules, introduced formally in the next chapter, are best understood as a specific, checkable mechanism for guaranteeing this property, rather than as a memory-safety feature that happens to have locality as a side effect.

4.4 Rust Mechanism: Bounding the Footprint at the Signature

Consider a function whose signature is `fn total(v: &Vec<i32>) -> i32`. To determine whether this function behaves correctly, a reader needs to know only what the function's body does with the reference it was given: it can read `v`, it cannot mutate it, and it cannot store it anywhere that outlives the call, since no lifetime has been threaded out through the return type. Nothing about the rest of the program, how many other places hold references to the same vector, whether some other thread might be running concurrently, whether some other function will later mutate the vector, is relevant to verifying `total`'s correctness. The reasoning footprint $N(\texttt{total}, v)$ is exactly the function's own signature and body, and it is bounded regardless of how large the surrounding program becomes.

This is not an accident of this particular example. It is the direct consequence of the aliasing guarantee that Chapter 5 will formalize: because a shared reference is guaranteed by the compiler not to be mutated by anything else for as long as it is held, a function that only receives shared references never needs to ask what else might be happening to the value concurrently. The guarantee is not merely convenient, it is exactly what converts an otherwise globally verifiable property, is this value stable for the duration of this read, into a locally verifiable one, checkable from the type signature alone.

```
fn total(v: &Vec<i32>) -> i32 {
    v.iter().sum()
    // Correctness of this function needs nothing beyond this
    // signature and this body. No other function in the program,
    // however large the program becomes, can invalidate this
    // reasoning, because &Vec<i32> already rules out concurrent
    // mutation for the duration of the call.
}
```

4.5 Worked Example: A Language Without the Guarantee

The value of the guarantee is easiest to see by imagining its absence. Suppose a hypothetical language allowed any function to read a shared reference while simultaneously permitting some other, arbitrary part of the program to mutate the same underlying value with no compile-time restriction. A function such as `total` above would then require a fundamentally different, and fundamentally more expensive, verification procedure: a reader would need to inspect every other function in the entire program that could conceivably hold a mutable path to the same vector, determine whether any of them could execute concurrently with `total`, and confirm that none of them do, for every call site, for the lifetime of the codebase.

```
// Hypothetical, not valid Rust:
// fn total(v: &Vec<i32>) -> i32 {
//     v.iter().sum()
//     // In a language without exclusivity guarantees, correctness
//     // here would depend on every other function that might hold
//     // a mutable alias to v -- an unbounded, whole-program search,
//     // repeated after every future change to the codebase.
// }
```

This is precisely the situation that arises in languages that permit unrestricted mutable aliasing, and it is why data races and iterator-invalidation bugs in such languages are notorious for surfacing far from their actual cause: the bug is not local to any one function, because the language never bounded any function's reasoning footprint in the first place.

4.6 Interpretation Exercise: Which Locality Guarantee Failed?

The following function compiles in Rust but is worth examining for what it reveals about the boundary of the locality guarantee, since not every function that compiles has a footprint as tightly bounded as `total`'s.

```
static mut COUNTER: i32 = 0;

fn record(delta: i32) {
    unsafe {
        COUNTER += delta;
    }
}
```

The question worth asking is not whether this compiles, since with an explicit `unsafe` block it does. It is: *which locality guarantee did the programmer have to give up to write it this way?* `record`'s signature, `fn record(delta: i32)`, gives no indication that the function touches anything beyond its argument; nothing in the type signature bounds its footprint to `COUNTER`. Verifying that two calls to `record` from different threads do not race requires inspecting every call site in the program, exactly the unbounded search that ownership and borrowing exist to avoid. The `unsafe` keyword is not incidental here; it is Rust's explicit acknowledgment that this function's reasoning footprint can no longer be constructed from its signature alone, and that the compiler is withdrawing its locality guarantee for this specific piece of code, by name, so that the loss is visible rather than silent.

Recurring Example: The Library System

A librarian processing a single checkout needs exactly one thing: the current record for the specific book being checked out. Nothing about correctly completing that transaction requires consulting the status of every other book in the catalog, the borrowing history of every other patron, or the schedule of every other branch. This is Section 4.2's locality principle in its most literal library form: the reasoning footprint $N(f, v)$ for a checkout operation is bounded by that one book's record, and a circulation system that somehow required scanning the entire catalog's state to process a single checkout would be exhibiting exactly the reasoning-cost failure Chapter 3's stale-belief example hinted at, now named directly. Chapter 5 picks this thread up and asks what it would take to guarantee, structurally, that a checkout can never accidentally depend on more than the one record it needs.

4.7 Connections: Locality Beyond Ownership

The value of establishing locality as its own principle, prior to ownership, is that it explains why ownership takes the particular shape it does rather than some other shape that would also prevent memory unsafety. A runtime garbage collector, for instance, also prevents use-after-free, but it does not restore locality of reasoning about concurrent mutation; two threads holding a reference to the same collected object can still race on it, because a collector's guarantee is about reachability for the purposes of deallocation, not about bounding any individual function's reasoning footprint. Ownership is a stronger, differently-shaped guarantee, and this chapter has argued it is stronger in exactly the dimension that matters for keeping large codebases tractable as they grow.

The same principle reappears well outside memory management. A well-designed trait boundary, covered in Part III, bounds a caller's reasoning footprint to the trait's contract rather than the implementing type's internals. A well-designed concurrent system, covered in Part V, bounds each thread's reasoning footprint to the messages it explicitly receives rather than the entire shared-memory state of the program. A well-designed module boundary, covered in Part VI, bounds a reader's reasoning footprint to a crate's public interface rather than its implementation. In every one of these cases, the underlying question is the same one posed in Section 4.2: is this property locally verifiable, or does it silently require the whole system to be held in mind at once?

This chapter's locality principle has a close relative in the philosophy of science. Mark Wilson's extended study of how physical and mathematical concepts are actually applied in practice argues that such concepts typically remain well-behaved only within bounded "patches," theories that work reliably in some local domain and quietly stop applying, or require patching together with a neighboring theory, once pushed past that domain's edge [1]. A physicist's classical-mechanics toolkit is not one seamless theory valid everywhere; it is a patchwork of locally reliable descriptions, each honest about its own boundary, stitched together where they meet. Chapter 4's reasoning footprint $N(f, v)$ is this same idea, translated into a claim about software: a function's correctness should be a locally verifiable patch, honest about its own boundary, rather than a claim that silently presupposes the rest of the program has been correctly modeled along with it.

Invariant Established in This Chapter

Correct reasoning about a component of a system should require only a bounded context, one that does not grow with the size of the surrounding system. A language, or a system design more generally, that fails to bound this context converts every local change into a potential global liability. The chapters that follow examine ownership, borrowing, and lifetimes as Rust's specific, compiler-checked mechanism for guaranteeing this bound at the level of individual values, and later chapters will show the same guarantee re-derived at the level of interfaces, threads, and modules.

Part I Summary

- Programs are state transformations, not transcripts of instructions (Chapter 1).
- Composition is only predictable when a program's declared state actually captures what it depends on (Chapter 2).
- Mutation invalidates the beliefs of any observer who reached a value beforehand (Chapter 3).
- Correct reasoning about a component should require only a bounded context (Chapter 4).

Therefore: a language that wishes to support scalable reasoning must make authority over state explicit. Part II begins the argument for what that explicit authority actually looks like.

Part II

Ownership as Reachability

Chapter 5: Ownership as Reachability

From First Principles

Question: Why can't two independent parts of a program freely mutate the same value?

5.1 Why Memory Management Is Fundamentally a Reachability Problem

Every program that runs for more than an instant accumulates state that outlives the instruction which created it. A number is written to memory so that a later instruction can read it; a buffer is allocated so that a later stage of a pipeline can fill it; a connection is opened so that a later message can be sent across it. In each case, something is being handed to the future. The central question underneath memory management is not, as introductory treatments often imply, "when should we free memory," but the earlier and more structural question: *who is currently able to reach this value, and what may they legitimately do with it once they reach it?*

Call the set of paths through which a value can be reached its *reachability set*. A local variable is reachable through its name in the enclosing scope. A field of a struct is reachable through the struct. An element behind a pointer is reachable through the pointer, and through every alias of that pointer. It is worth fixing this as an object of study rather than leaving it as a metaphor, since the rest of the chapter keeps returning to it: for a value v, write

$$R(v) = \{\, p \mid p \text{ is a program path capable of accessing } v \,\}.$$

Memory errors, in nearly every language that has ever produced one, are failures of reachability accounting: a value is freed while $R(v)$ is still non-empty (use-after-free), a value is freed twice because two paths in $R(v)$ both believed themselves the sole owner (double-free), or two paths in $R(v)$ act on the value at the same time with incompatible intentions, one reading while the other writes underneath it (a data race).

Garbage-collected languages solve part of this problem by tracking reachability at runtime and refusing to reclaim anything still reachable. This is a real solution to the deallocation half of the problem, but it says nothing about the second half: it does nothing to prevent two reachability paths from mutating a value simultaneously in incompatible ways, because reachability alone does not distinguish observation from mutation, or shared access from exclusive access. A garbage collector answers "is this value still reachable at all," not "is this particular access to this value currently admissible." Rust's ownership model is best understood as an attempt to answer the second, harder question at compile time, for both memory safety and logical safety simultaneously, without paying the runtime cost of a collector.

5.2 Objects as Resources with Admissible Continuations

Take any value in a running program: an integer, a vector, a file handle, a lock. At any moment, that value's history so far constrains what can legitimately happen to it next. A file that has been closed cannot admissibly be read from again. A vector that has been moved into another function cannot admissibly be read from in the function that moved it. A lock that is currently held cannot admissibly be acquired again by the same thread without deadlocking. In each case, the value's past determines a set of *admissible continuations*: the operations that can validly extend its history without producing an inconsistency. Formally, if $\mathcal{O}$ is the set of operations that could in principle be performed on v, then at a given state s of v's history there is a subset

$$\mathcal{C}(v, s) \subseteq \mathcal{O}$$

of operations currently permitted. Most of what follows in this chapter can be read as commentary on how Rust computes $\mathcal{C}(v, s)$ at compile time and rejects any program that performs an operation outside it.

This is the same idea that appears throughout the reconstruction and repair literature under the name *admissibility*: not every transition from a given state is a legitimate one, and a system that fails to track which transitions are legitimate will eventually be asked to perform an illegitimate one, silently, with no local signal that anything has gone wrong. A language that does not track admissible continuations at the type level pushes the entire burden of tracking them onto the programmer's memory and discipline. This works until it doesn't, and when it fails, it fails at runtime, often far from the site of the actual error, which is why memory bugs in C and C++ are notorious for manifesting nowhere near their cause.

Rust's insight, inherited from earlier work on linear and affine type systems, is that a resource's admissible continuations can often be fully determined by tracking one thing: how many places, and what kind of places, currently have the authority to act on it. This single piece of bookkeeping, enforced at compile time, turns out to be enough to rule out the entire class of reachability failures described in the previous section, without a runtime collector and without runtime locks in the common case.

What Goes Wrong Without This: Double-Free

Before stating the rule, it is worth seeing the specific failure it exists to rule out. Consider a language that lets two variables independently believe they own the same heap allocation, with no compiler tracking which belief is still valid:

```
// Hypothetical, not valid Rust:
// let a = allocate_buffer();
// let b = a;              // b now also "owns" the same allocation
// free(a);                // a's owner frees it
// free(b);                // b's owner, unaware a already freed it, frees
    it again
// -- the allocator's internal bookkeeping is now corrupted, and the
// -- effects of this corruption may not surface until an entirely
// -- unrelated allocation fails, far from this code.
```

Neither line individually looks wrong; each simply frees what it was told it owns. The bug is that the language never disqualified `a` from still believing it was an owner after `b` was

created, exactly the diffusion of authority Section 5.1 warned about. A double-free is not a rare, exotic failure; it is the direct, predictable consequence of a language having no mechanism to track how many paths currently believe themselves entitled to decide a value's fate.

5.3 Ownership as Uniqueness of Authority

Rust's ownership rule can be stated compactly: every value has exactly one owner at a time, and when that owner goes out of scope, the value's resources are released. What this rule is actually doing, underneath the compiler-rule phrasing, is guaranteeing that responsibility for a value's continuation is never diffused. At every point in the program's execution, there is exactly one place with the authority to decide what happens to a given value next: to mutate it, to move it elsewhere, or to let it end.

This is why assignment in Rust moves rather than copies by default for types that manage a resource. When `let b = a;` is written for such a type, treating both `a` and `b` as valid owners afterward would immediately violate uniqueness of authority; two places would each believe themselves entitled to decide the value's fate, and whichever runs first would silently invalidate the other's assumption. Rust's response is to make the old path, `a`, inadmissible after the move. The compiler does not merely warn about this. It refuses to compile any subsequent use of `a`, because such a use is not a stylistic mistake, it is a reference to a continuation that the type system can prove is no longer authorized to exist.

```
let a = String::from("reachable");
let b = a;           // ownership moves to b
// println!("{a}"); // compile error: a's continuation ended at the move
println!("{b}");     // b is the sole admissible continuation
```

Seen this way, the move is not an arbitrary restriction bolted onto the language for safety's sake. It is the direct, minimal consequence of taking seriously the claim that a resource's continuation should have exactly one author at a time. Every other ownership-related rule in the language is a variation on the same theme, applied to progressively more flexible situations.

The same uniqueness shows up constantly outside software, usually enforced by a physical or procedural mechanism rather than a compiler. A library book checked out to one borrower cannot simultaneously be checked out to a second borrower; the library's own records make the first borrower the sole party currently authorized to decide what happens to the book next, and a second patron wanting it must wait for that authority to be released. A rental car has exactly one active renter at a time, and the rental agency's paperwork is the mechanism enforcing it. A master key held by a single security officer, a relay race baton passed hand to hand with never more than one runner holding it, a signed package requiring an explicit custody transfer at each handoff, a token in a board game occupying exactly one square and controlled by exactly one player's decisions at a time, all encode the identical rule: authority over what happens next is unique, and transferring it is an explicit, visible act, not a quiet duplication.

What Goes Wrong Without This: Iterator Invalidation

A second failure, subtler than a double-free, motivates the distinction the rest of this section develops. Consider iterating over a collection while another part of the program is permitted to resize it mid-iteration:

```
// Hypothetical, not valid Rust:
// for item in &collection {
//     if some_condition(item) {
//         collection.push(new_item); // resizes the backing storage
//     }
//     // the iterator's internal cursor may now point past the end
//     // of a freshly reallocated buffer, or into memory the old
//     // buffer no longer occupies at all
// }
```

The iterator was handed a promise, implicitly, that the collection would not change shape while it was being read; nothing in a language without Chapter 5's aliasing discipline enforces that promise, and violating it produces behavior ranging from skipped elements to a crash, depending on how the collection happens to be implemented underneath. The failure is not the mutation itself, mutation is often exactly what a program needs to do; it is that nothing marked the iterator's read as exclusive of concurrent mutation, precisely the coexisting-authority problem Section 5.1 introduced in the abstract.

5.4 Borrowing as Temporary Admissible Observation

Requiring unique ownership for every access would make Rust nearly unusable; a function that merely wants to read a value would have to take ownership of it, consume it, and hand it back explicitly every time. Most access to a value is not a request to redirect its continuation at all. It is a request to *observe* it: to read its current state without altering the fact that a specific owner is still the one responsible for what happens to it next.

A shared reference, written `&T`, is exactly this: temporary, non-exclusive, admissible observation. Holding a `&T` does not transfer ownership and does not entitle the holder to mutate the value; it only entitles the holder to read it, for as long as the reference remains valid. Because observation does not change what the value's next legitimate continuation could be, Rust permits an arbitrary number of shared references to the same value to coexist. Ten readers looking at an unchanging value cannot produce an inconsistency among themselves, because none of them is authorized to alter what the others are seeing.

It is worth separating two things a reference can grant, since the rest of the chapter depends on keeping them apart: *authority*, the right to determine v's next admissible continuation, and *knowledge*, the right to observe v's current state. A shared reference grants knowledge without authority. This is not a weaker version of ownership; it is a different kind of relationship to the value entirely, and it is why an arbitrary number of shared references can coexist safely while at most one authority-granting reference can exist at a time. The distinction reappears, under different names, wherever a system separates an observer from whatever is entitled to intervene on the thing being observed.

Everyday knowledge-without-authority relationships are easy to find once the distinction is named. Looking at a museum exhibit behind glass grants any number of simultaneous visitors full knowledge of what is on display without granting any of them authority to rearrange it. Reading a newspaper gives the reader complete knowledge of its contents while leaving authority over the next day's edition entirely with the publisher. Borrowing someone's paper map to check a route lets the borrower observe it fully without the map's owner losing anything, and without the borrower gaining any right to redraw it. Watching security camera

footage, live or recorded, gives any number of simultaneous viewers full knowledge of what the camera saw, with authority over the camera and its feed remaining exactly where it was.

```
fn describe(v: &Vec<i32>) -> usize {
    v.len() // read-only: an admissible observation, not a continuation
        change
}

let data = vec![1, 2, 3];
let a = &data;
let b = &data;
println!("{} {}", describe(a), describe(b)); // many observers, one owner
```

5.5 Mutable Borrowing as Exclusive Authority

Mutation is a different kind of act than observation. To mutate a value is to redirect its continuation, even if only temporarily and even if ownership itself is not being transferred. If two independent parts of a program were permitted to mutate the same value at the same moment, each would be silently invalidating assumptions the other is relying on; this is precisely the aliasing hazard that Part I's discussion of mutation set up in the abstract, now made concrete.

A mutable reference, `&mut T`, is Rust's mechanism for granting temporary, exclusive authority over a value without transferring ownership outright. The compiler's aliasing rule, that a value may have either any number of shared references or exactly one mutable reference, but never both at once, is not an arbitrary restriction for its own sake. It is the direct consequence of the same admissibility reasoning applied to mutation: an act that changes what continuations are legitimate for a value must not coexist with any other act that is relying on the value's current state remaining fixed.

Exclusive, temporary authority of exactly this shape is a familiar pattern well outside programming. A mechanic servicing heavy machinery under a lockout-tagout procedure applies a personal lock and a tag bearing their own name to the power switch before beginning work, and every other worker's attempt to restart the machine has to physically contend with that lock, not a posted sign or a verbal warning, for exactly as long as the mechanic's own tag remains on it; the tag does not merely block the switch, it names, visibly, who currently holds the authority the lock represents. A surgeon operating on a patient holds sole authority over what happens to the patient's body during the operation, even while observers watch through glass; the observers retain knowledge, in Section 5.4's sense, but none of the authority the surgeon alone holds. A bank teller accessing a cash drawer that is locked to everyone else for the duration of that transaction is granted exclusive authority over the drawer's contents, released the moment the transaction ends and the drawer is locked again. In each case, exclusivity is temporary and scoped tightly to the duration of the act that actually requires it, exactly as `&mut T` is scoped to the span during which it is held.

```
fn increment(v: &mut Vec<i32>) {
    v.push(0); // exclusive authority: safe to mutate
}

let mut data = vec![1, 2, 3];
increment(&mut data);
```

```
// while the &mut is active, no other reference, shared or mutable, may
   exist
```

This is why the compiler rejects code that holds a shared reference across a mutation, even when the mutation happens to be harmless in practice. The compiler cannot see into the future to confirm harmlessness; it can only verify, structurally, that no reachability path exists through which an observer's assumptions could be invalidated out from under it. Rejecting all cases where such a path *could* exist, rather than trying to prove which specific cases would actually go wrong, is what makes the guarantee checkable at compile time instead of requiring runtime detection.

Consider the following program, which the compiler refuses to build:

```
let mut v = vec![1, 2, 3];
let r = &v;
v.push(4);
println!("{r:?}");
```

The question worth asking is not simply "why doesn't this compile." It is: *which admissibility claim became impossible for the compiler to certify?* The reference `r` was granted knowledge of `v`'s state at a moment when that state consisted of three elements with a fixed backing allocation. The call to `v.push(4)` is a request for exclusive authority, which may in general require reallocating `v`'s backing storage entirely, invalidating any prior knowledge-granting reference to it. Once that authority is exercised, the compiler can no longer certify that `r` still refers to anything meaningful, so it withdraws `r`'s admissibility retroactively, at the point the mutation occurs, rather than waiting to discover at runtime whether this particular reallocation happened to occur.

What Goes Wrong Without This: Dangling Reference

A third failure motivates lifetimes specifically, as distinct from ownership and borrowing: a reference that outlives the value it points to.

```
// Hypothetical, not valid Rust:
// fn dangling() -> &i32 {
//     let local = 42;
//     &local  // local's storage is reclaimed when this function returns
// }
// let r = dangling();
// println!("{r}"); // reads whatever now occupies that reclaimed memory
```

`r` was handed a reference to a location that stopped being valid the moment `dangling` returned; reading it afterward is reading memory the language has already reused for something else, silently, with no indication at the read site that anything is wrong. The bug is not that a reference was returned; it is that nothing checked whether the reference's claimed validity span actually fit within its referent's real one, exactly the gap Section 5.1's reachability accounting exists to close.

5.6 Lifetimes as Proofs That Observations Remain Valid

A reference is only meaningful for as long as the value it points to remains alive and in the state the reference's holder expects. A reference that outlives its referent, a so-called dangling

reference, is a reachability path pointing at nothing, or worse, at memory that has since been reused for something else entirely. Lifetimes are Rust's mechanism for making the *duration* over which an observation remains admissible into something the compiler can check, rather than something the programmer must remember.

A lifetime annotation such as `'a` in `fn longest<'a>(x: &'a str, y: &'a str) -> &'a str` is not introducing a new kind of runtime value. It is a compile-time proof obligation: whatever the function returns must not be permitted to outlive whichever of its inputs it was actually derived from. The compiler is not tracking time in the way a scheduler would; it is tracking a partial order of admissible validity spans and rejecting any program that would let a reference's claimed span exceed the actual span its referent guarantees.

Time-bounded authorizations of exactly this kind are common enough outside software that the underlying logic is usually taken for granted. A visitor pass expires at the end of the day, after which whatever access it granted becomes inadmissible regardless of whether the visitor still has the physical badge in hand. A temporary parking permit is valid only for its stated window; presenting it after that window has closed does not extend the authorization, it simply fails to establish one. A hotel key card is deliberately programmed to stop working at checkout, converting an implicit social expectation, guests leave when their stay ends, into a mechanically enforced validity span. A construction permit carries an expiration date past which the authorized work is no longer legally covered by it, however unfinished the work remains. In each case, the authorization's validity was always bounded; the expiration does not take something away so much as it makes visible a limit that was part of the grant from the beginning, exactly as a lifetime does not shorten a reference's usefulness but states, honestly, how long it was ever going to be valid.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
// The signature is a promise: the returned reference's validity
// cannot exceed the shorter-lived of the two inputs' validity spans.
```

Read this way, lifetimes are not a separate, harder topic bolted onto ownership and borrowing. They are the temporal component of the same admissibility question the chapter opened with: not only *who* may currently reach a value and *what* they may do with it, but *for how long* that authorization remains valid. A borrow checker error about a lifetime is, at bottom, always the same complaint: a piece of code is claiming a continuation that the type system cannot prove was ever actually granted.

5.6.1 Elision: When the Proof Is Obvious

Most functions that take and return references in real Rust code carry no explicit lifetime annotation at all, which can make lifetimes feel like a rule that mysteriously applies only sometimes. What is actually happening is that the compiler is capable of constructing the validity-span proof itself in a large class of common cases, and only asks the programmer to supply the annotation when the proof would otherwise be ambiguous.

```
fn first_word(s: &str) -> &str {
    s.split_whitespace().next().unwrap_or("")
}
// No annotation needed: there is exactly one reference parameter,
```

```
// so the only candidate proof is that the output's span cannot
// exceed the input's span. The compiler fills this in silently.
```

This case, one input reference and one output reference, is unambiguous enough that the compiler infers the proof automatically; this is one instance of Rust's lifetime elision rules. The `longest` function from earlier has two candidate input spans, `x` and `y`, and no rule can determine from the signature alone which one the return value's span should be tied to, since that depends on which branch executes at runtime. An explicit `'a` annotation in a case like this is not extra syntax layered on top of the proof; it is the proof itself, supplied by the programmer because the compiler has no unique way to construct it on its own. Elision, in other words, is not a separate feature from lifetimes; it is what happens whenever the proof obligation has only one possible discharge, and the annotation only resurfaces once more than one discharge becomes possible.

5.7 The Borrow Checker as an Admissibility Verifier

It is worth being explicit about what the borrow checker is, because most introductions leave this implicit and let it sit in the reader's mind as an obstacle to be worked around. The borrow checker is not a linter, and it is not enforcing a style preference. It is a static verifier for a specific, checkable claim: that every reachability path present in the compiled program corresponds to an authorization the ownership and borrowing rules actually grant, for exactly as long as the program claims to hold it.

This is precisely analogous to a repair-warrant check in the reconstruction and repair sense: a claim that a given transition is legitimate is only accepted once it has been verified against the constraints that define legitimacy for that kind of transition, rather than being taken on the programmer's word. The borrow checker's error messages are frequently experienced as friction because they are refusing to compile code whose eventual runtime behavior might, in a specific execution, have been harmless. But the checker is not verifying a specific execution; it is verifying the whole space of executions the program's structure permits, and refusing to certify any program for which that space contains even one inadmissible transition.

5.8 Idiomatic Examples

The following short program exercises ownership, shared borrowing, and mutable borrowing together, and is worth reading with the reachability vocabulary of this chapter actively in mind rather than as a syntax exercise.

```
struct Ledger {
    entries: Vec<i64>,
}

impl Ledger {
    fn new() -> Self {
        Ledger { entries: Vec::new() }
    }

    fn record(&mut self, amount: i64) {
```

```
        self.entries.push(amount); // exclusive authority required to
            mutate
    }

    fn balance(&self) -> i64 {
        self.entries.iter().sum() // shared observation is sufficient to
            read
    }
}

fn main() {
    let mut ledger = Ledger::new();   // ledger is the sole owner of its
        entries
    ledger.record(100);
    ledger.record(-40);

    let total = ledger.balance();     // a temporary shared borrow, then
        released
    println!("balance: {total}");

    let handle = &mut ledger;         // exclusive authority granted to
        handle
    handle.record(25);                // ledger itself cannot be used here
    println!("balance: {}", ledger.balance()); // handle's authority has
        ended
}
```

Every line in `main` is doing exactly one of three things: transferring ownership, granting a temporary shared observation, or granting temporary exclusive authority. Once a reader can identify which of the three is happening at each line without consulting the compiler, they have internalized the actual content of the ownership system, independent of any particular syntax.

5.9 Exercises

5.9.1 Write a function that takes ownership of a `String`, appends a suffix, and returns it. Explain, in reachability terms rather than compiler terms, why the caller's original variable becomes unusable after the call.

5.9.2 Modify the `Ledger` example so that two functions attempt to hold a `&mut Ledger` at the same time. Predict the compiler's objection before compiling, in terms of exclusive authority, then confirm it.

5.9.3 Write a function with a lifetime-annotated signature that returns a reference into one of two input slices depending on a runtime condition. Explain what the annotation is proving, and what would go wrong if the language allowed the function to be written without it.

5.9.4 Rewrite the `Ledger` example using `Rc<RefCell<Vec<i64>>>` to permit shared mutable access from multiple owners. Identify precisely which of this chapter's compile-time guarantees have been traded for a runtime check, and why that trade might be worth making in a specific context.

A Practical Extension: Shared and Runtime-Checked Reachability

Every guarantee this chapter has developed so far depends on the compiler being able to determine, from the program's static structure, how many paths in $R(v)$ exist and what each is entitled to do. Some designs, a graph whose nodes are shared between several parents, a cache several unrelated parts of a program all need to update, do not have a shape the compiler can fully see in advance; the number of owners a node should have is a runtime fact, not something derivable from the code's structure alone. Rust does not abandon Section 5.3's uniqueness-of-authority principle for these cases; it relocates its verification from compile time to runtime, using the same overall discipline this book will later generalize in Chapter 21's treatment of `unsafe`: the underlying obligation does not disappear, only the mechanism that checks it changes.

`Rc<T>` ("reference counted") permits multiple owners of the same value, tracking the count of active owners at runtime and releasing the value only once that count reaches zero; it is Section 5.3's ownership rule with "exactly one owner" relaxed to "at least one owner, counted precisely." `RefCell<T>` performs the analogous relaxation for Section 5.5's aliasing rule: rather than the compiler statically proving that no `&mut` coexists with a `&`, a `RefCell` tracks borrows at runtime and panics, immediately and loudly, the moment a violation is attempted, exactly the admissible-manifold departure Chapter 15 will formalize. The aliasing rule itself has not weakened, only where it is checked.

```
use std::cell::RefCell;
use std::rc::Rc;

let ledger = Rc::new(RefCell::new(vec![100, -40]));
let ledger2 = Rc::clone(&ledger); // a second owner, reference count now 2

ledger.borrow_mut().push(25);      // runtime-checked exclusive borrow
println!("{:?}", ledger2.borrow()); // runtime-checked shared borrow, same
    data
```

A third type completes the family. `Rc<T>` alone can produce reference cycles, two nodes each owning a strong reference to the other, that never reach a count of zero and therefore never release, a memory leak that Section 5.3's compile-time discipline would have made structurally impossible but that a runtime reference count cannot detect on its own. `Weak<T>` is a non-owning reference, one that does not keep the count above zero and must be explicitly upgraded, fallibly, back into a strong reference before use; a parent-to-child relationship typically uses `Rc`, and the corresponding child-to-parent back-reference typically uses `Weak`, breaking the cycle by making one direction of the relationship deliberately non-authoritative. This is worth noticing as a design choice, not a language quirk: it is Section 5.4's authority-versus-knowledge distinction, applied to a cyclic structure, so that only one direction of a mutual reference counts as ownership at all.

Recurring Example: The Library System

Chapters 1 through 4 followed a library's checkout system far enough to motivate this chapter's central question: what would it take to guarantee, structurally, that the stale-belief failure of Chapter 3 and the bounded-context requirement of Chapter 4 both hold by construction. Ownership answers it directly. A checked-out book has exactly one current borrower, in Sec-

tion 5.3's sense: the moment it is checked out, authority over what happens to it next, renewing the loan, returning it, reporting it lost, belongs to that one borrower's record and no other, exactly the uniqueness Section 3.7's two librarians needed and did not have.

```
struct Book { title: String, status: LoanStatus }
struct Loan { book: Book, borrower_id: u32 } // the book's authority moves
    here

fn check_out(book: Book, borrower_id: u32) -> Loan {
    Loan { book, borrower_id } // ownership of `book` moves into the Loan
}
```

A librarian pulling up a record to answer "is this available" is doing what Section 5.4 called borrowing: temporary, non-exclusive knowledge of the book's current status, granted without disturbing whoever currently holds authority over it, exactly the way any number of patrons can check the online catalog's listing for a book simultaneously without any of them gaining the right to check it out on someone else's behalf. And the loan itself has a lifetime in Section 5.6's sense: a due date bounding how long the borrower's authority remains valid, checked, in a real library, the way a borrow checker validates a reference, against records the system keeps of when that authority was actually granted.

5.10 Connections to Larger Architectures

The reasoning developed in this chapter does not stay confined to single-threaded, single-owner programs. When a value is shared across threads, the same question, who may currently reach this value and what may they legitimately do with it, becomes a concurrency question rather than a single-threaded memory question, and Part V returns to ownership under exactly that lens: a channel send is a move of ownership across a thread boundary, and a mutex guard is a runtime-checked mutable borrow whose exclusivity is enforced at the moment of acquisition rather than at compile time.

The same reasoning also appears directly in the author's own Rust projects. `physiome`'s pure functional core relies on ownership to guarantee that a simulated physiological state cannot be mutated by two subsystems in the same tick without that mutation being visible at the type level; the admissibility and repair vocabulary used throughout that project's design is not a metaphor imported after the fact, it is the same structural claim this chapter has been making about ownership itself, applied one level up, to whole simulation states rather than individual values. A reader who has internalized this chapter is already equipped to read that codebase's structure as an extension of the same idea, not as a new one.

Ownership, in the end, is one instance of a much broader pattern rather than a Rust-specific peculiarity. Large systems in general become easier to reason about when authority, observation, and continuation are made explicit rather than left implicit and reconstructed from memory or convention. Rust applies this principle to the memory of a single running program. The remaining parts of this book will show the same principle reappearing at larger scales: in concurrency, where the question becomes which thread currently holds authority over a shared value; in asynchronous execution, where continuation itself becomes an explicit, suspendable object; in interpreters and distributed systems, where the same reachability and admissibility questions apply to values that may not even reside on the same machine; and

in software architecture generally, where module boundaries are, at bottom, decisions about who is granted authority over what.

It is worth being explicit about a point this chapter has been implicitly relying on throughout: a constraint of the kind Rust's ownership rule imposes is not merely a restriction subtracted from what a programmer could otherwise do. Alicia Juarrero's account of constraint in complex systems argues that context-dependent constraints are frequently generative rather than merely limiting, that removing a degree of freedom at one level can be exactly what creates new, previously unavailable possibilities at another [2]. Fearless concurrency, the subject of Chapter 17, is the clearest instance of this in the present book: the ownership rule subtracts a freedom, the freedom to alias a mutable value from more than one path, and that subtraction is precisely what makes safe, checked concurrent programming possible at all. The constraint does not merely prevent a class of bugs; it is the enabling condition for a class of programs, safely shared mutable state managed without a garbage collector, that would not be reachable without it.

Invariant Established in This Chapter

Every safe Rust program preserves uniqueness of authority over each resource, $|\{p \in R(v) : p \text{ currently holds authority over } v\}| \leq 1$, while permitting arbitrarily many concurrent non-authoritative observations of that resource. Ownership, borrowing, mutable borrowing, and lifetimes are not four independent rules to be memorized separately. They are the mechanisms by which this single invariant is established, checked, and maintained across every execution path a program's structure admits.

Part II Summary

- Every value has exactly one owner at a time; authority over its continuation is never diffused (Chapter 5).
- Observation and authority are distinct: any number of shared observers may coexist, but only one path may hold authority to mutate (Chapter 5).
- A reference's validity is bounded in time by a lifetime, a checkable proof rather than a convention (Chapter 5).
- The borrow checker is a static verifier for exactly these claims, not a style preference (Chapter 5).

Therefore: ownership is not an isolated language feature but the mechanism by which Part I's bounded local reasoning is actually maintained over mutable, shared state. Parts III through VI extend this same mechanism to composite data, partiality, concurrency, and every boundary a program eventually crosses.

Part III

Composition Before Inheritance

Chapter 6: Structs and the Shape of State

From First Principles

Question: When you declare the shape of a value, are you describing what it *is*, or constraining what it is *allowed to become*?

6.1 Phenomenon: Forms That Rule Things Out

A passport application form has named fields: given name, date of birth, place of birth, photograph. The form does not merely provide convenient boxes to write in; it declares, by omission, that a passport application does not include a blood type, a favorite color, or a childhood address, because no such field exists to hold them. Anyone auditing a filled-out application knows exactly what to expect to find in it, and exactly what to expect *not* to find, before reading a single filled-in value. The form's shape is doing real work: it is bounding the space of things an application can possibly be.

An electrical outlet's physical shape does the same. A standard household socket accepts a plug of a specific geometry and rejects, by simple non-fit, anything shaped differently. Nobody needs to inspect the current flowing through a device to know it cannot be plugged in incorrectly; the shape of the socket has already ruled out the inadmissible configurations before any current flows at all. The constraint is enforced structurally, by the geometry, rather than procedurally, by someone checking each time.

Standardized paper forms of every kind work the same way as the passport application. A medical record's fixed fields, patient identifier, diagnosis codes, prescribed treatment, declare by omission that a chart is not the place a clinician's personal opinion of the patient belongs, because no field exists to hold it. A shipping label's fixed layout, sender, recipient, weight, tracking number, declares exactly what a courier needs and nothing else, and a label missing any of those fields is not a valid label, regardless of how much other information might be written on the package. A tax form's line items declare, precisely, which figures the filing authority considers relevant to a return, ruling out any financial detail the form provides no line for. A library catalogue card's fields, title, author, call number, declare the complete shape of what the catalogue considers a book's identity, independent of anything else that might be true about the book. In every case, the form is not a suggestion about what is usually written down; it is a declaration of what the record could possibly be.

A struct declaration in a programming language is doing the same kind of work, though it

is easy to read it as mere bookkeeping, a convenient way to bundle related values together. A struct with three named fields of specific types is not simply offering a convenient container; like the passport form and the outlet, it is declaring, by its shape, the complete space of values that could ever count as an instance of it, and ruling out, by construction, everything else.

6.2 General Principle: A Type Declares a State Space

Chapter 1 identified a program with the transition it realizes over some state space Σ. That chapter left Σ mostly unexamined, treating it as a background set of possible states. This chapter asks where the shape of Σ actually comes from for a composite value, one built out of several smaller pieces, rather than treating that shape as given. The answer is that the shape is declared, deliberately and in advance, by whoever defines the type, and every value of that type is thereafter constrained to fit within the declared shape.

This is a stronger claim than it might first appear. It is not merely that a struct is convenient notation for grouping related data. It is that the struct definition is itself the complete specification of what states are admissible for values of that type; a value that does not conform to the declared shape, one field short, one field extra, one field holding a value outside its declared type's own admissible range, is not a variant of the type. It is not a member of the type's state space at all. The passport form and the electrical socket worked the same way: the shape was not descriptive of what typically shows up, it was constitutive of what could possibly count as valid.

6.3 Formal Definition: Structs as Product State Spaces

If a struct is declared with fields of types $T_1, T_2, \ldots, T_n$, its state space is the Cartesian product

$$\Sigma_{\text{struct}} = \Sigma_{T_1} \times \Sigma_{T_2} \times \cdots \times \Sigma_{T_n},$$

where Σ_{T_i} is the admissible state space of the i-th field's own type. Every legal instance of the struct corresponds to exactly one tuple in this product, and, critically, every tuple in this product corresponds to some legal instance: nothing is declared in the fields that cannot appear, and nothing can appear that was not declared in the fields. The struct's admissible continuations, in the sense introduced in Chapter 5, are exactly those operations that produce a new tuple still lying within Σ_{struct}; there is no mechanism, short of `unsafe` code deliberately circumventing it, for a struct's value to ever leave this product space.

This is worth contrasting with a more permissive alternative, familiar from dynamically typed or loosely structured languages: a record represented as an open map from field names to values, where any key can be added, removed, or left absent at runtime. Such a record's state space is not a fixed product at all; it is closer to the power set of all possible key-value pairs, and any code reading such a record must be prepared for keys it did not expect and the absence of keys it did. A Rust struct forecloses this entire category of uncertainty at the type level, by declaring the product in advance and enforcing it at every construction site.

6.4 Rust Mechanism: Construction as the Only Entry Point

```
struct Coordinate {
    latitude: f64,
    longitude: f64,
}
```

There is exactly one way to bring a `Coordinate` into existence when all fields are public and no custom constructor is used: supply a value for every field, of the declared type, in a single struct-literal expression. There is no partially constructed `Coordinate`, no `Coordinate` missing a `longitude`, and no `Coordinate` whose `latitude` field holds a `String`. The compiler will not compile a program that attempts any of these, which means the guarantee from Section 6.3, that every value of the type lies in $\Sigma_{T_1} \times \Sigma_{T_2}$, does not need to be checked anywhere else in the program; it is established once, at the type's boundary, and every later piece of code that receives a `Coordinate` can simply rely on it.

```
let home = Coordinate { latitude: 40.7128, longitude: -74.0060 };
// home is guaranteed, by construction, to have both fields present
// and both fields holding valid f64 values. No code anywhere else
// in the program needs to re-check this.
```

6.5 Worked Example: Encoding a Narrower State Space Than the Product Suggests

Not every constraint a programmer wants can be expressed by the product structure alone. Suppose latitude must lie between -90 and 90 degrees. The raw product $\Sigma_{\texttt{f64}} \times \Sigma_{\texttt{f64}}$ admits latitudes far outside that range, so the struct's declared shape, on its own, is broader than the space of values the programmer actually intends to be admissible.

```
struct Coordinate {
    latitude: f64,
    longitude: f64,
}

impl Coordinate {
    fn new(latitude: f64, longitude: f64) -> Option<Coordinate> {
        if (-90.0..=90.0).contains(&latitude) {
            Some(Coordinate { latitude, longitude })
        } else {
            None
        }
    }
}
```

Making the fields private and routing construction exclusively through `new` narrows the type's *effective* state space back down to the intended subset of the product, by refusing to construct anything outside it in the first place. This is the same pattern Section 6.2 described in the abstract, applied deliberately: the struct's declared shape sets an upper bound on what is representable, and a guarded constructor can narrow that bound further, but nothing in ordinary, safe Rust can ever widen it back out.

Historical Note: The Mars Climate Orbiter

In 1999, NASA's Mars Climate Orbiter was lost on approach to Mars because one software module, built by a contractor, produced thrust data in pound-seconds, while the navigation software receiving that data expected newton-seconds. Both values were ordinary floating-point numbers; nothing in either system's types distinguished a pound-second from a newton-second, so the mismatch produced no error at the boundary where the two modules exchanged data, only a trajectory correction off by a factor of roughly 4.45, enough to send the spacecraft fatally close to the planet's atmosphere.

This is exactly the failure Section 6.3 formalized: a bare `f64` representing a thrust value has a state space, $\Sigma_{\texttt{f64}}$, that says nothing about which unit the number is denominated in, and two modules exchanging bare `f64`s are trusting a convention neither type enforces. A struct-based alternative closes the gap directly:

```
struct NewtonSeconds(f64);
struct PoundSeconds(f64);

impl From<PoundSeconds> for NewtonSeconds {
    fn from(p: PoundSeconds) -> Self {
        NewtonSeconds(p.0 * 4.44822)
    }
}

fn apply_correction(thrust: NewtonSeconds) { /* ... */ }
// apply_correction(PoundSeconds(100.0)); // does not compile: wrong type
   entirely
```

With thrust values wrapped in distinct, single-field structs rather than left as bare `f64`s, Chapter 6's product-state-space guarantee does real work: a `PoundSeconds` value can no longer be passed anywhere a `NewtonSeconds` is expected without an explicit, visible conversion, converting a silent unit mismatch that cost a spacecraft into an ordinary type error caught before the program ever runs.

6.6 Interpretation Exercise: A Shape That Promises More Than It Delivers

```
pub struct Coordinate {
    pub latitude: f64,
    pub longitude: f64,
}

impl Coordinate {
    pub fn new(latitude: f64, longitude: f64) -> Option<Coordinate> {
        if (-90.0..=90.0).contains(&latitude) {
            Some(Coordinate { latitude, longitude })
        } else {
            None
        }
    }
}
```

```
fn nudge_north(c: &mut Coordinate) {
    c.latitude += 1.0;
}
```

This compiles. The question worth asking is: *which claim from Section 6.5 has quietly stopped being true?* Making the fields `pub` reopened direct field access alongside the guarded constructor, so `nudge_north` can push `latitude` to 91.0 without ever passing back through `new`'s check. The struct's nominal shape, $\Sigma_{\texttt{f64}} \times \Sigma_{\texttt{f64}}$, was never actually narrowed at the type level in the first place; only the *constructor* was narrowed, and any code with direct field access can bypass a constructor entirely. The lesson is not that guarded constructors are useless, but that the narrowing they provide is only as strong as the visibility rules that force every code path through them; a single `pub` field is enough to reopen the full, unconstrained product space Section 6.3 described.

Recurring Example: The Library System

The library's book record, informally assumed throughout Chapters 1 through 5, finally gets a concrete shape:

```
struct Book {
    title: String,
    author: String,
    isbn: String,
}
```

This is Section 6.3's product state space made literal: a `Book` is exactly a title, an author, and an ISBN, together, no more and no less, and every value of type `Book` is guaranteed, by construction, to have all three. Nothing about a book's checkout status belongs in this struct; that is a fact about the book's *current state*, not about its identity, and Chapter 7 gives that distinction its own type.

Connection to Category Theory

Section 6.3's product state space, $\Sigma_{T_1} \times \cdots \times \Sigma_{T_n}$, is a categorical *product*: the universal construction of an object equipped with a projection to each T_i and carrying no information beyond what those projections jointly determine. Rust's structs and tuples are, in this light, direct syntax for a construction category theory studies in far greater generality [9].

6.7 Connections: Structs as the First of Several Shape-Declaring Mechanisms

This chapter has treated structs as a way of declaring a product state space, one whose admissible values are exactly the combinations of its declared fields. Chapter 7 turns to enums, which declare a state space by *sum* rather than by product, a value being exactly one of several named alternatives rather than all fields at once, and which handle a kind of state Chapter 6's product structure cannot express cleanly: mutually exclusive possibilities. Chapter 9 turns to

traits, which declare a state space's *behavior* rather than its data layout, specifying what a type can do without specifying what it is made of. All three are variations on the same question this chapter opened with: what shape is being declared, and what does that shape rule out.

Invariant Established in This Chapter

A value's admissible state space is exactly what its declared type specifies, no more and no less. A struct's fields fix a product space of admissible combinations, established once at the type's boundary rather than re-checked at every later use; narrowing that space further, to enforce constraints the field types alone cannot express, requires routing every construction path through a guarded constructor and withholding direct field access, since a single unguarded path back into the type is enough to restore the full, unconstrained shape the type's fields originally declared.

Chapter 7: Enums and Explicit Branching

From First Principles

Question: If a value can be one of several different things, where should the list of possibilities actually live?

7.1 Phenomenon: Possibilities Left Implicit

A traffic light is red, yellow, or green, never some blend of the three and never a fourth color. The set of possible states is small, fixed, and known in advance to every driver approaching the intersection, which is exactly why a traffic light can coordinate behavior among strangers who have never met: everyone is reasoning about the same closed list of possibilities, and no driver needs to wonder whether some undocumented fourth state might appear. A traffic light that could, in principle, ever show an undocumented color would be useless as a coordination device, however rarely that color actually appeared.

A jury's verdict form typically allows for exactly two outcomes, guilty or not guilty, and nothing else; there is no blank line for a third option, because the legal process has decided in advance that no third option is admissible. The form does not describe the range of ambiguity jurors might feel while deliberating; it forces their deliberation to resolve into one of exactly two declared outcomes before the process can proceed. The ambiguity is real during deliberation, but the form refuses to let it survive into the result.

Closed lists of mutually exclusive possibilities are everywhere once the pattern is named. A chess piece is a pawn, knight, bishop, rook, queen, or king, never something in between and never a seventh kind of piece invented mid-game. A currency's denominations, whatever coins and notes a country issues, form a fixed, closed list that every price and every transaction must resolve into, with no room for a denomination nobody minted. A deck of playing cards assigns every card to exactly one of four suits, and a card belonging to two suits at once, or to none, is not a malformed card, it is not a card at all. An operating system's process states, running, waiting, sleeping, terminated, are enumerated exhaustively by the scheduler precisely because a process the scheduler cannot classify into one of its known states is a process the scheduler cannot correctly manage.

A great deal of software instead represents multi-way possibilities the way an informal note might: a status field that is usually one of a few strings, an integer code whose meaning is documented only in a comment, a combination of a boolean flag and a nullable pointer that

together are supposed to mean one of several things depending on which combination shows up. In each of these cases, the list of possibilities exists, but it exists only in the programmer's head, or in a comment, or in documentation, rather than anywhere the compiler can see it. Chapter 6 asked where the shape of a composite value's state space comes from and answered: the struct declaration. This chapter asks the parallel question for a value that is one of several mutually exclusive alternatives, rather than all of several fields at once.

What Goes Wrong Without This: Invalid Sentinel Values

Before the closed-list discipline is stated formally, it is worth seeing the failure mode it replaces: encoding a possibility as a special value of an otherwise ordinary type, rather than as its own named case.

```
// Hypothetical, not valid Rust:
// fn find_index(items: &[i32], target: i32) -> i32 {
//     for (i, &item) in items.iter().enumerate() {
//         if item == target { return i as i32; }
//     }
//     -1 // sentinel meaning "not found"
// }
// let idx = find_index(&data, 7);
// let next = data[idx as usize + 1]; // idx == -1 silently underflows
   here
```

`-1` is not a real index; it is an ordinary `i32` being asked to double as a signal that no answer exists, and nothing in the type `i32` distinguishes a genuine index from the sentinel standing in for absence. A caller who forgets to check for `-1` before using the result does not get an error, since `-1` type-checks as fine wherever an `i32` was expected; the mistake surfaces later, if at all, as a wrong or out-of-bounds access far from the point where the absence actually occurred. The failure is exactly the one Chapter 12 will later name directly, a claimed domain, "any `i32`, presumably a valid index," that silently exceeds the honest domain the function actually intended.

7.2 General Principle: A Closed List, Declared Once

Chapter 6 described a struct as declaring a state space by product: a value of the type must supply all of its fields simultaneously. Many real values are not naturally described this way. A network request is either still pending, or it has succeeded with some payload, or it has failed with some error, but it is never simultaneously pending and succeeded; the three possibilities are mutually exclusive, not simultaneous. A state space for such a value should be declared by *sum*, not by product: a value is exactly one alternative out of a fixed, named list, never more than one and never something outside the list.

Where a possibility is left implicit, encoded by convention in a magic number, a status string, or a combination of flags whose joint meaning lives only in a comment, the checking of that possibility is left implicit too. Every piece of code that inspects the value must independently know, and correctly re-derive, which combinations are meaningful and which are not, and nothing prevents a new piece of code from producing a combination nobody anticipated.

Declaring the list once, in a place every consumer of the value is required to consult, converts a fact that used to live in documentation into a fact the compiler can enforce.

7.3 Formal Definition: Enums as Sum State Spaces

If an enum is declared with variants $V_1, V_2, \ldots, V_n$, where variant V_i carries an associated payload of type T_i (possibly the unit type, for a variant carrying no data), its state space is the disjoint union, or *sum*,

$$\Sigma_{\text{enum}} = \Sigma_{T_1} \sqcup \Sigma_{T_2} \sqcup \cdots \sqcup \Sigma_{T_n}.$$

A value of the enum's type belongs to exactly one summand at a time, tagged by which variant produced it; unlike the struct's product from Chapter 6, there is no state in which more than one variant's data is simultaneously present, and no state in which none is. This is the type-level counterpart of the traffic light and the verdict form: the list of possibilities, V_1 through V_n, is closed and declared once, and every value of the type is guaranteed, by construction, to be tagged as exactly one of them.

The practical consequence follows directly from the disjointness of the sum. Because Σ_{enum} has no summand corresponding to "none of the above" and no summand corresponding to "more than one at once," any code that wishes to inspect an enum value must, to be exhaustive, account for every V_i; there is no valid value that could fall through a check that covers all of them, and this is precisely what Rust's `match` exhaustiveness checking, covered in Section 7.4, is able to verify mechanically.

7.4 Rust Mechanism: Exhaustiveness as a Compiler-Checked Guarantee

```
enum RequestState {
    Pending,
    Succeeded(String),
    Failed(String),
}
```

A function that matches on a `RequestState` is required, by the compiler, to handle all three variants, or to include an explicit catch-all arm acknowledging that some variants are being intentionally ignored. There is no way to write a `match` on `RequestState` that silently forgets `Failed`; the omission is a compile error, not a latent bug waiting to be discovered at runtime when a request happens to fail for the first time in production.

```
fn describe(state: &RequestState) -> String {
    match state {
        RequestState::Pending => "waiting".to_string(),
        RequestState::Succeeded(body) => format!("ok: {body}"),
        RequestState::Failed(err) => format!("error: {err}"),
    }
    // Every variant of Σ_enum is accounted for. If a fourth
    // variant were added to RequestState tomorrow, this function
    // would fail to compile until it decided what to do with it.
}
```

7.5 Worked Example: A Closed List That Grows Safely

The value of exhaustiveness checking is easiest to see under change. Suppose `RequestState` later gains a fourth variant, `Cancelled`.

```
enum RequestState {
    Pending,
    Succeeded(String),
    Failed(String),
    Cancelled,
}
```

Every existing `match` expression over `RequestState` in the entire codebase, however large, now fails to compile until it is updated to account for `Cancelled`. This is not the compiler being obstructive; it is the compiler enforcing, at every consumption site simultaneously, the same guarantee Section 7.3 described: the list of possibilities is closed, and it has just changed, so every place that claimed to handle the complete list must now be shown to still be telling the truth. A status field encoded as a string or an integer code offers no such guarantee; adding a new status value silently leaves every existing check unaware that a new possibility now exists, and the resulting bug typically surfaces only once the new status value actually occurs at runtime, far from the site where it was introduced.

7.6 Interpretation Exercise: Which Possibilities Did the Catch-All Hide?

```
fn is_done(state: &RequestState) -> bool {
    match state {
        RequestState::Succeeded(_) => true,
        _ => false,
    }
}
```

This compiles under both the three-variant and the four-variant version of `RequestState`. The question worth asking is: *what did the catch-all arm, `_ => false`, actually decide, and did it decide it correctly for every variant it swallowed?* Under the three-variant version, the catch-all covers `Pending` and `Failed`, both of which are indeed not done in the sense the function name suggests, so the collapse is harmless. Once `Cancelled` is added, the same catch-all silently absorbs it too, reporting a cancelled request as not done, which may or may not be the intended meaning of `is_done`; a cancelled request is, in an important sense, also finished. The exhaustiveness guarantee from Section 7.4 only extends as far as the arms actually named; a catch-all arm is a deliberate, and sometimes dangerous, opt-out of exhaustiveness, since it will silently absorb any future variant without forcing the author to reconsider whether the absorption is still correct.

Recurring Example: The Library System

Chapter 6 gave a book its identity; this chapter gives it a status, and the status is exactly the kind of closed, mutually exclusive list Section 7.3 formalized:

```
enum LoanStatus {
```

```
    Available,
    CheckedOut { borrower_id: u32, due: Date },
    Reserved { patron_id: u32 },
    Lost,
}
```

A book is in exactly one of these four states at a time, never available and checked out simultaneously, never lost and reserved at once, and the compiler enforces this the moment `LoanStatus` is declared this way, exactly as Section 7.2 argued a closed list should be. Chapter 8 picks up the question of what a librarian's software is entitled to assume once it has determined which of these four states a given book is actually in.

Connection to Category Theory

Section 7.3's disjoint-sum state space is the categorical dual of Chapter 6's product: a *coproduct*, the universal construction of an object built from any one of several injections, with no way to hold more than one at once. Structs and enums are, in this light, two halves of the same categorical picture, and Rust's type system happens to give both first-class syntax [9].

7.7 Connections: Sums, Products, and What Comes Next

Chapter 6 gave composite values a way to declare a state space by product; this chapter has given them a way to declare a state space by sum. Real data is very often a combination of both: a struct whose fields are themselves enums, or an enum whose variants carry struct payloads, builds arbitrarily rich state spaces out of these two primitives alone. Chapter 8 turns to pattern matching in more depth, treating a `match` expression not merely as a branching construct but as the reconstruction of which summand of a sum type a given value currently occupies, an idea that will recur, transformed, when Part IV reaches `Option` and `Result`, both of which are, structurally, nothing more than small enums whose variants happen to represent presence and absence, or success and failure, rather than the traffic-light-style states used as this chapter's examples.

Invariant Established in This Chapter

Every continuation is explicit. When a value may be one of several mutually exclusive alternatives, the complete list of alternatives should be declared once, in the type, rather than left to be reconstructed independently by every piece of code that inspects the value. An enum's state space is the disjoint sum of its variants; the compiler's exhaustiveness check is what converts "I have considered every possibility" from a claim the programmer hopes is true into a claim the compiler has verified is true, and re-verifies automatically every time the list of possibilities changes.

Chapter 8: Pattern Matching as Reconstruction

From First Principles

Question: When you ask "which case is this," are you discovering something about the value, or are you constructing a claim that still needs to be justified?

8.1 Phenomenon: Diagnosis as Reconstruction, Not Discovery

A doctor examining a patient with a fixed list of candidate diagnoses is not passively reading a label already attached to the patient. The doctor is actively reconstructing, from symptoms, test results, and history, which one of the candidate diagnoses the evidence actually supports, and a competent doctor does not stop after checking the first plausible candidate; the process is only trustworthy once every candidate on the list has been weighed and ruled in or out. A diagnosis reached by checking one possibility and assuming the rest do not apply is not a diagnosis, it is a guess wearing a diagnosis's clothing.

An auditor reconciling a ledger against a fixed list of permissible transaction categories is doing the same kind of work. Each entry must be reconstructed as belonging to exactly one category, and an entry that does not obviously fit any of them is not silently filed under the closest-looking one; it is flagged, because forcing an ill-fitting entry into the wrong category would corrupt the reconciliation while looking, from the outside, exactly like a completed one.

Chapter 7 established that an enum's state space is a closed sum of named alternatives, and that the compiler can verify a `match` expression accounts for all of them. This chapter looks more closely at what a `match` expression is actually doing at the moment it runs, since Chapter 7 focused on the shape of the type being matched rather than on the act of matching itself. The doctor and the auditor suggest the answer: matching is not reading a label that was already there, it is reconstructing, from a value's runtime tag, which member of a known, closed list of alternatives that value currently is.

8.2 General Principle: Matching Recovers a Fact That Construction Established

Chapter 6 and Chapter 7 both located the guarantee about a value's shape at construction time: a struct's fields are fixed once at the moment it is built, and an enum's variant is fixed

once at the moment it is built. Neither guarantee would be worth much if it could not later be recovered. Pattern matching is the mechanism by which a piece of code, holding a value of a sum type, recovers the specific fact that construction established: which variant, out of the closed list, this particular value actually is.

This framing matters because it clarifies what a `match` arm is actually claiming. An arm is not an independent check the programmer invented on the spot; it is a claim that, if the value's runtime tag matches this pattern, then the payload bound by the pattern is guaranteed, by the type's own construction discipline from Chapter 7, to have the shape the pattern expects. The guarantee was established once, at construction; matching is where that guarantee gets cashed in, and it is only as trustworthy as the exhaustiveness that Chapter 7 showed the compiler enforces across every arm simultaneously.

8.3 Formal Definition: Matching as a Case Analysis Over a Sum

Recall $\Sigma_{\text{enum}} = \Sigma_{T_1} \sqcup \Sigma_{T_2} \sqcup \cdots \sqcup \Sigma_{T_n}$ from Chapter 7. A `match` expression over a value $v \in \Sigma_{\text{enum}}$ with arms $A_1, \ldots, A_n$, one per variant, is a case analysis: it partitions the reasoning about v into n independent sub-arguments, one per summand, where sub-argument A_i is permitted to assume $v \in \Sigma_{T_i}$ and nothing more. The expression as a whole is sound exactly when every summand of Σ_{enum} is covered by some arm, which is precisely the exhaustiveness property Chapter 7 introduced, now viewed from the perspective of the case analysis it licenses rather than from the perspective of the type declaration that made it checkable.

This is worth stating formally because it explains why a partial match, one that does not cover every summand, is unsound in a specific, technical sense rather than merely incomplete in spirit: a partial match's uncovered arms correspond to a portion of Σ_{enum} for which no sub-argument has been supplied at all. Rust's refusal to compile a non-exhaustive `match` without an explicit catch-all is a refusal to accept a case analysis with a gap in its domain.

8.4 Rust Mechanism: Destructuring as Extraction Under a Verified Assumption

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
}

fn area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle { radius } => std::f64::consts::PI * radius * radius
            ,
        Shape::Rectangle { width, height } => width * height,
    }
}
```

Inside the `Circle` arm, the name `radius` is bound to an `f64` without any further check, because the arm's pattern, `Shape::Circle { radius }`, has already discharged the obligation

of confirming that this particular `Shape` value actually carries a `radius` field of that type; Chapter 6's construction guarantee for the payload of the `Circle` variant is what licenses skipping any further validation here. The `Rectangle` arm makes the same move for a different, disjoint region of Σ_{Shape}. Neither arm needs to consider the other's payload shape, because the case analysis from Section 8.3 has already separated them.

8.5 Worked Example: Nested Reconstruction

```
enum Shape {
    Circle { radius: f64 },
    Rectangle { width: f64, height: f64 },
}

enum Command {
    Draw(Shape),
    Erase,
}

fn describe(cmd: &Command) -> String {
    match cmd {
        Command::Draw(Shape::Circle { radius }) =>
            format!("draw circle r={radius}"),
        Command::Draw(Shape::Rectangle { width, height }) =>
            format!("draw rect {width}x{height}"),
        Command::Erase => "erase".to_string(),
    }
}
```

Here the reconstruction happens two levels deep in a single pattern: the first level recovers which variant of `Command` is present, and, within the `Draw` arm's payload, the second level simultaneously recovers which variant of `Shape` it wraps. The compiler still requires exhaustiveness across the full nested space; the three arms shown correspond exactly to the three reachable combinations, `Draw(Circle)`, `Draw(Rectangle)`, and `Erase`, and no combination of the two enums' variants is left unaccounted for.

Historical Note: Apple's "goto fail"

In 2014, a widely used implementation of TLS certificate validation in Apple's operating systems shipped with a duplicated line. The verification routine, stripped to its structural essentials, looked like this:

```
// C, reconstructed from the public disclosure, not Rust:
// if ((err = SSLHashSHA1.update(...)) != 0)
//     goto fail;
// if ((err = SSLHashSHA1.update(...)) != 0)
//     goto fail;
//     goto fail;                          // the extra line
// if ((err = SSLHashSHA1.final(...)) != 0)
//     goto fail;
// ...
```

```
// fail:
//     return err;
```

The second `goto fail;` sits on its own line, outside the `if` above it, so it executes unconditionally every time control reaches it, before the final signature check ever runs. Because `err` was still `0` at that point, from the last successful step, the function returned success regardless of whether the certificate the skipped check would have examined was valid or forged. The intended admissible continuation was `hash` → `verify signature` → `accept certificate`; what the duplicated line actually produced was `hash` → `goto fail` → `accept certificate`, a transition that was supposed to require successful signature verification and, because of one misplaced jump, no longer did. In this book's vocabulary, the admissible continuation set was silently enlarged: a state, "accept this certificate," that should only have been reachable after every check passed became reachable without satisfying the invariant the checks existed to establish, and the code implementing the final check became unreachable, dead code the compiler had no reason to flag.

This is a failure of unstructured control flow specifically, and it is worth noting what Chapter 8's reconstruction discipline rules out about it directly: Rust has no `goto` at all, and a `match` expression's arms are selected by which variant a value actually is, in Section 8.3's case-analysis sense, never by falling through a sequence of jumps that could be duplicated or reordered by a stray extra line. A validation routine expressed as a `match` over an explicit `enum ValidationStep` could not silently skip its final arm the way the duplicated `goto` did; every arm is reached by reconstruction of the value's actual variant, not by a control-flow path a careless edit could accidentally extend or truncate. The bug was never a failure of cryptography; it was a failure to preserve the intended transition structure, and Chapter 2 returns to this same example from the composition side, showing what it looks like when one stage of a verification pipeline is silently skipped rather than merely reached incorrectly.

8.6 Interpretation Exercise: A Reconstruction That Assumes Too Much

```rust
fn describe_v2(cmd: &Command) -> String {
    if let Command::Draw(shape) = cmd {
        match shape {
            Shape::Circle { radius } => format!("draw circle r={radius}"),
            _ => "draw other".to_string(),
        }
    } else {
        "not drawing".to_string()
    }
}
```

This compiles. The question worth asking is: *what portion of the reconstruction from Section 8.3 has quietly been left unjustified?* The inner `match` covers `Circle` explicitly and absorbs `Rectangle` into a catch-all labeled `"draw other"`, exactly the kind of opt-out Chapter 7 warned about. If a third `Shape` variant is added later, this function will continue to compile and will silently describe it as `"draw other"` without anyone having decided whether that description is correct for the new variant. Section 8.2's claim, that a match arm cashes in a guarantee established at construction, only holds for the arms actually written out by name; a catch-all

arm reconstructs nothing, it merely declines to ask the question, and the burden of correctness for whatever it silently absorbs has been quietly transferred from the compiler back onto whichever human remembers, unprompted, to revisit it.

Recurring Example: The Library System

Deciding what a checkout desk is allowed to do with a given book is a direct application of Section 8.2's reconstruction: the software must recover which of Chapter 7's four `LoanStatus` variants the book is actually in before it can act, and Chapter 7's exhaustiveness guarantee ensures no fifth, unanticipated status can slip through unhandled.

```
fn attempt_checkout(status: &LoanStatus, patron_id: u32) -> Result<(),
    String> {
     match status {
         LoanStatus::Available => Ok(()),
         LoanStatus::CheckedOut { .. } => Err("already checked out".into())
             ,
         LoanStatus::Reserved { patron_id: p } if *p == patron_id => Ok(())
             ,
         LoanStatus::Reserved { .. } => Err("reserved for another patron".
             into()),
         LoanStatus::Lost => Err("copy is lost".into()),
     }
}
```

Every arm reconstructs a specific, previously-established fact about the book, not a guess; the `Reserved` case even reconstructs *whose* reservation it is, licensing the comparison against `patron_id` only because the pattern has already confirmed the variant carries that field. A catch-all arm here, absorbing `Reserved` and `Lost` into one generic "cannot check out" message, would repeat Section 8.6's warning: it would satisfy exhaustiveness while discarding exactly the distinction a patron calling to ask why their checkout failed would want answered.

8.7 Connections: Reconstruction as a Recurring Theme

The pattern established in this chapter, that verifying a case analysis is exhaustive is what makes it trustworthy, reappears throughout the rest of this book under different names. `Option`, in Part IV, is a two-variant enum whose `None` case is exactly the kind of possibility a catch-all arm is tempted to paper over, and Rust's refusal to let a value be used without first matching on it is this chapter's exhaustiveness guarantee applied to the single most common source of null-related bugs in other languages. `Result`, likewise, forces an error variant to be reconstructed and handled rather than silently absorbed. The reconstruction and repair literature this book draws on throughout uses this same principle at a much larger scale, treating the recovery of a system's actual state from partial or degraded information as a case analysis that is only trustworthy once it can be shown to be exhaustive over the space of what could have happened; a `match` expression is, in miniature, the same commitment.

Invariant Established in This Chapter

Pattern matching reconstructs which admissible alternative a value currently occupies; it does not discover a label that was passively sitting on the value, it recovers a fact that the type's own construction discipline already established and licenses each arm to assume. This reconstruction is only trustworthy when it is exhaustive: a catch-all arm does not perform reconstruction for the cases it swallows, it defers the question, and any variant added later will pass silently through it without anyone having verified that the deferral is still correct.

Chapter 9: Traits as Interface Contracts

From First Principles

Question: When you promise that something can be done, are you promising *how*, or only promising *that*?

9.1 Phenomenon: Contracts That Hide Their Implementation

A shipping contract between a retailer and a carrier specifies a small number of observable commitments: a package meeting certain size and weight limits will arrive at its destination within a stated number of days, in intact condition, for a stated price. The contract says nothing about which trucks the carrier uses, which routes its drivers take, or which warehouse the package passes through at two in the morning. The retailer does not need any of that information to rely on the contract; the retailer needs only to know that whatever the carrier does internally, the observable commitments will be honored.

An electrical appliance and a wall outlet share a contract of exactly this shape. The outlet promises a certain voltage, at a certain frequency, through a certain physical connector shape. It does not promise, and the appliance does not need to know, which power plant generated the electricity, which substation stepped it down, or which route the wiring took through the walls. Chapter 6 showed that a struct's shape rules out what a value's internal state can be; this chapter is about a different kind of shape, one that rules out nothing about internal state at all and instead constrains only what a type can be observed to *do*.

Behavioral contracts of this kind govern a wide range of everyday interfaces. A USB port specifies a fixed electrical and data protocol that any compliant device can speak, regardless of whether the device behind the connector is a keyboard, a storage drive, or a webcam; the port's contract says nothing about, and does not care, what the device is internally, only that it honors the interface. An electrical plug and socket standard works the same way for power rather than data. A musical instrument is playable if it can be made to produce controlled pitch and rhythm on demand, a contract a violin and a synthesizer satisfy through entirely different physical mechanisms. A checkout counter's accepted payment methods, cash, card, mobile wallet, are each required only to honor the contract "transfer this amount of value to the merchant," with the internal mechanics of a bank transfer and a folded banknote having nothing in common beyond satisfying that one shared promise. And the distinction between what a category of thing typically does and what a specific contract actually requires is itself instructive: not every bird can fly, and some fish, and no birds at all, can swim, so a contract like "can fly" or "can swim" is best stated as its own promise, satisfied by whichever specific

types actually satisfy it, rather than assumed from a broader category membership that does not, in fact, guarantee it.

Software interfaces exist to make exactly this kind of promise available between pieces of code that do not know, and should not need to know, about each other's internals. A function that accepts anything implementing a `Serialize` interface does not need to know whether the concrete type behind it is a struct, an enum, or something else entirely; it needs only to know that the observable operation, serialization, is available and behaves according to the interface's contract.

What Goes Wrong Without This: Duplicated Code

Before traits are introduced formally, it is worth seeing what a codebase looks like without any mechanism for stating a shared contract once. Suppose a program needs to print a human-readable description of several unrelated types, and the language offers no way to declare "anything describable" as a single concept:

```
// Hypothetical, not valid Rust:
// fn print_circle_description(c: &Circle) { println!("circle, radius {}",
    c.radius); }
// fn print_square_description(s: &Square) { println!("square, side {}", s
   .side); }
// fn print_triangle_description(t: &Triangle) { println!("triangle, base
   {} height {}", t.base, t.height); }
// -- every new shape requires its own, separately named print function,
// -- and every caller that wants to print "whatever shape this is"
// -- needs its own copy of the same branching logic to pick which one to
   call.
```

Each function is fine in isolation; the failure is systemic. A bug fixed in `print_circle_description`'s formatting has to be separately remembered and reapplied to `print_square_description` and `print_triangle_description`, since nothing connects them as instances of one underlying contract. Worse, a function that wants to accept "anything printable" has no way to say so; it must either accept a fixed, closed list of concrete types, or duplicate its own logic once per type, multiplying the maintenance burden every time a new shape is added. Chapter 9 is not a convenience for avoiding repetitive typing; it is the mechanism that lets "anything satisfying this contract" be stated once, as a real, checkable claim, rather than reconstructed by hand at every call site.

9.2 General Principle: Behavior Without Representation

Chapters 6 through 8 were concerned with a type's *representation*: what data it holds, and in what shape. A trait is concerned with a type's *behavior*: what operations can be performed on it, described entirely in terms of their signatures, independent of how any particular type chooses to implement them. Two types with completely different representations, one a struct wrapping a single number, another an enum with several variants, can implement the same trait identically as far as any caller of the trait's methods is concerned, provided both honor the same observable contract.

This is the distinguishability-preserving move the trait system makes possible: a caller that depends only on a trait, rather than on a concrete type, has deliberately discarded information about representation in exchange for the ability to work with any type that honors the contract, including types that do not yet exist at the time the caller was written. The cost of this flexibility is that the caller can rely on nothing beyond what the trait's contract actually states; anything true of one implementation but not guaranteed by the trait itself is not available to code that only knows the value through the trait.

9.3 Formal Definition: A Trait as a Behavioral Admissibility Contract

A trait $\mathcal{T}$ declares a set of method signatures $\{m_1, m_2, \ldots, m_k\}$, each specifying an operation's inputs and outputs, without specifying an implementation. A type S *implements* $\mathcal{T}$ by supplying a concrete definition for every m_i that type-checks against its declared signature. Crucially, the trait itself defines the complete set of admissible operations that code depending only on $\mathcal{T}$, rather than on S directly, is entitled to perform: for any value known only by its trait bound, the admissible continuations available to the caller, in the sense of Chapter 5, are exactly $\{m_1, \ldots, m_k\}$, regardless of what additional operations S itself might separately support.

This is why a trait is best understood as an *admissibility contract* rather than merely a grouping of related functions. It does not describe everything S can do; it describes everything that can be relied upon by a caller that has deliberately chosen to know S only through $\mathcal{T}$. The gap between what S can actually do and what $\mathcal{T}$ promises is not a defect; it is the entire point, since that gap is what allows a second type S', with a completely different full set of capabilities, to be substituted wherever only $\mathcal{T}$'s contract is required.

9.4 Rust Mechanism: Trait Bounds as the Caller's Declared Dependency

```
trait Describe {
    fn describe(&self) -> String;
}

struct Circle { radius: f64 }
struct Square { side: f64 }

impl Describe for Circle {
    fn describe(&self) -> String {
        format!("circle, radius {}", self.radius)
    }
}

impl Describe for Square {
    fn describe(&self) -> String {
        format!("square, side {}", self.side)
    }
}
```

```
fn print_description(item: &impl Describe) {
    println!("{}", item.describe());
    // This function's entire dependency on `item` is the single
    // method declared by Describe. It cannot access `radius` or
    // `side` directly, because neither is part of the contract.
}
```

`print_description` can be called with a `Circle`, a `Square`, or any future type that implements `Describe`, without being modified. This is Chapter 4's locality principle reappearing in a new form: the function's reasoning footprint is bounded by the trait's contract, not by the total space of types that might someday implement it.

9.5 Worked Example: Substitutability in Practice

```
struct Triangle { base: f64, height: f64 }

impl Describe for Triangle {
    fn describe(&self) -> String {
        format!("triangle, base {} height {}", self.base, self.height)
    }
}

fn describe_all(items: &[&dyn Describe]) {
    for item in items {
        println!("{}", item.describe());
    }
}

let c = Circle { radius: 2.0 };
let s = Square { side: 3.0 };
let t = Triangle { base: 4.0, height: 5.0 };
describe_all(&[&c, &s, &t]);
```

`describe_all` was written before `Triangle` existed in this example's narrative, yet it accepts `Triangle` without modification, because `Triangle`'s only obligation was to honor `Describe`'s contract. This is the payoff of Section 9.3's admissibility-contract framing made concrete: the function depends on the contract, not on the closed list of types available at the time it was written, in direct contrast to Chapter 7's enums, whose closed list of variants is exhaustively checked precisely because it is *not* meant to be open to arbitrary future extension in the same way.

9.6 Interpretation Exercise: A Contract That Promises Less Than It Seems To

```
trait Describe {
    fn describe(&self) -> String;
}
```

```
fn compare_lengths(a: &impl Describe, b: &impl Describe) -> bool {
    a.describe().len() > b.describe().len()
    // compiles
}
```

This compiles and runs, comparing the lengths of the two description strings. The question worth asking is: *does this function's behavior actually rely on anything `Describe` promises, or has it smuggled in an assumption the contract never made?* `Describe` promises only that `describe` returns a `String`; it says nothing about that string's length being meaningful for comparison, nothing about it being stable across calls, and nothing about it correlating with any property of the underlying shape. `compare_lengths` type-checks because `String` supports `.len()`, but its actual usefulness depends on an assumption, that description length tracks something meaningful, that lives entirely outside the trait's stated contract. This is the trait-level analogue of Section 6.6's unguarded `pub` field: the type system enforces the contract exactly as declared, and any reliance on more than the contract states is a silent, unverified assumption the compiler has no way to catch.

Recurring Example: The Library System

A real library catalogs more than books: DVDs, journals, sheet music, each with its own record shape, but a shared checkout desk that needs to treat all of them uniformly wherever they behave alike. Traits express exactly this, in Section 9.3's admissibility-contract sense:

```
trait Borrowable {
    fn check_out(&mut self, patron_id: u32) -> Result<(), String>;
}

trait Reservable {
    fn reserve(&mut self, patron_id: u32) -> Result<(), String>;
}
```

A `Book` might implement both; a rare reference-only journal might implement neither, only a `Searchable` trait for the catalog; a DVD might implement `Borrowable` but not `Reservable` if the library's policy does not allow holds on media. A function written to accept anything `Borrowable` can process a checkout for any of these types without knowing, or needing to know, what else the concrete type can or cannot do, exactly Section 9.4's caller-declares-its-own-dependency discipline: the checkout desk's software depends on `Borrowable` alone, not on "being a Book."

9.7 Connections: Interfaces as Locality for Behavior

Chapter 4 established locality as a general principle: correct reasoning about a component should require only a bounded context. Chapters 5 through 8 showed ownership, borrowing, structs, and enums each enforcing a version of that principle for a value's data. This chapter has shown traits enforcing the same principle for a value's behavior: a function written against a trait bound has a reasoning footprint fixed by the trait's declared methods, independent of how many types come to implement it or how large the surrounding codebase grows. Chapter 10 extends this further with generics, which let a function depend on a trait bound without

committing to any single concrete type at all, and Chapter 11 examines why composing several narrow trait contracts together tends to preserve more of this locality than inheriting from a single broad base type would.

Invariant Established in This Chapter

Interfaces preserve behavioral admissibility. A trait declares the complete set of operations that code depending only on the trait, rather than on a concrete type, is entitled to rely upon; this is deliberately narrower than everything an implementing type can do, and that narrowness is what allows unrelated types to be substituted for one another wherever only the trait's contract is required. Any behavior relied upon beyond what the trait actually states is an unverified assumption, not a guarantee the type system has checked.

Chapter 10: Generics and Parametric Admissibility

From First Principles

Question: If you can prove a claim without knowing what you are talking about, what does that tell you about the claim?

10.1 Phenomenon: Proofs That Do Not Need to Know the Subject

A mathematician proving that "for any set X, if $f : X \to X$ is injective and X is finite, then f is also surjective" does not need to know whether X is a set of numbers, a set of chess positions, or a set of sandwiches. The proof goes through using only the stated properties, injectivity and finiteness, and nothing else about X's specific nature. This is not a weakness of the proof; it is the source of its power. Because the argument never leaned on any fact specific to numbers, it applies equally to chess positions and sandwiches, without needing to be re-derived for each.

A moving company that promises to transport "any item under 500 kilograms that fits through a standard doorway" has written a policy that does not need to enumerate pianos, refrigerators, and bookshelves separately. The policy's guarantee holds for anything satisfying the two stated constraints, whatever that thing turns out to be, including items the company has never been asked to move before. Chapter 9 showed a trait bound letting a function depend on a contract rather than a concrete type. This chapter asks what happens when that idea is pushed one step further: not a function written for one contract-bound type, but a function, or a whole data structure, written for *any* type meeting a stated set of constraints, with the constraints themselves doing all the work the mathematician's injectivity and finiteness did.

Containers designed around a stated constraint, rather than around a specific cargo, are common well outside mathematics. A standardized shipping container is built to a fixed set of exterior dimensions and structural load limits, and satisfies its purpose for furniture, machine parts, or produce alike, without the container's own design needing to change based on which of these it happens to carry on a given voyage. A bookshelf built to hold items within a stated height and weight range serves novels, textbooks, or binders equally well, because the shelf's design constraint was stated in terms of size and weight, not in terms of "books" specifically. A filing cabinet's drawers, sized for a standard paper format, hold legal documents, blank forms, or old correspondence with no need for a different cabinet design per document type. A toolbox with a fixed set of slot sizes accepts any tool that fits those slots, screwdrivers, wrenches, pliers, without the box needing a separate compartment designed

around each specific tool. In every case, the container's usefulness comes from stating a constraint precisely enough that anything satisfying it is automatically accommodated, exactly the move a generic function makes when it is bounded by a trait rather than written against one concrete type.

What Goes Wrong Without This: Copy-and-Paste Implementations

Chapter 9 showed what happens without any way to state a shared behavioral contract. A related but distinct failure shows up even once traits exist, if there is still no way to write a single function that works across every type satisfying one: the function itself has to be copy-pasted, once per concrete type, with only the type name changed.

```
// Hypothetical, not valid Rust without generics:
// fn largest_i32(items: &[i32]) -> i32 { /* ... */ }
// fn largest_f64(items: &[f64]) -> f64 { /* ... */ }
// fn largest_temperature(items: &[Temperature]) -> Temperature { /* ...
   */ }
// -- three functions, identical in every line except the type name,
// -- each requiring its own bug fixes, its own tests, and its own
// -- eventual drift as small inconsistencies creep in between copies.
```

The three functions are not really three different pieces of logic; they are one piece of logic, typed three times. A bug found in `largest_i32` does not automatically get fixed in `largest_f64`, and a fourth type added later requires writing, and independently verifying, a fourth near-identical copy. Chapter 10's generics exist precisely to let this one piece of logic be written, and verified, exactly once, with the compiler responsible for confirming it as valid for every type meeting the stated bound, rather than the programmer responsible for keeping several manually-duplicated copies in sync by hand.

10.2 General Principle: Deferring the Choice of Type Without Deferring the Guarantee

A function written for a single concrete type makes a claim about that one type. A function written generically over a type parameter T, constrained by some set of trait bounds, makes a claim about every type that could ever satisfy those bounds, including types that do not exist yet at the time the function is written. This is a stronger claim, not a vaguer one: because the function's body can only use what the bounds guarantee, and nothing about any specific T, the compiler can verify the claim once, for the shape of the bound, rather than needing to re-verify it separately for every concrete type that eventually instantiates it.

This is worth contrasting with a superficially similar but weaker alternative: writing a function against a very general, loosely-typed input, such as "anything at all," and hoping the operations used inside happen to be defined for whatever arrives at runtime. That alternative defers the check to the moment of use, and it can fail for some values of the general type and not others. A properly bounded generic defers only the *choice* of type, never the guarantee; the bound is checked once, at the function's definition, against every operation the body performs, and any concrete type that cannot satisfy the bound is rejected before the function can even be called with it.

10.3 Formal Definition: Parametric Admissibility

Let $\mathcal{T}$ be a trait, and let $\mathcal{U}$ be the (possibly infinite, possibly not-yet-fully-known) set of types that implement $\mathcal{T}$. A function generic over a type parameter $T : \mathcal{T}$ is a single definition that, once verified, holds simultaneously for every $S \in \mathcal{U}$:

$$\forall\, S \in \mathcal{U}, \quad f_S \text{ is well-typed and behaves according to the same proof that verified } f_T.$$

Call this property *parametric admissibility*: the function's correctness is established once, over the bound, and inherited automatically by every type that later satisfies the bound, without any of those types needing to be known, or even to exist, at verification time. This is the formal counterpart to the mathematician's proof from Section 10.1: the proof was verified once, over the stated properties, and applies to every X satisfying them, discovered or undiscovered.

Note the relationship to Chapter 9's admissibility contract: a trait bound restricts a generic function's admissible continuations to exactly the trait's declared methods, in precisely the sense Chapter 9 defined, but here that restriction is what makes the single verification valid across an entire, possibly-open-ended family of types, rather than for one type alone.

10.4 Rust Mechanism: Bounds as the Complete Set of Available Operations

```
use std::fmt::Display;

fn largest<T: PartialOrd + Copy>(items: &[T]) -> T {
    let mut max = items[0];
    for &item in items.iter() {
        if item > max {
            max = item;
        }
    }
    max
}
```

Inside `largest`, the only operations available on a value of type `T` are those guaranteed by `PartialOrd` (comparison, via `>`) and `Copy` (duplication without consuming the original). The compiler verifies this function exactly once, checking that its body never asks for an operation the bounds do not supply, and that single verification then covers `i32`, `f64`, a custom `Temperature` type implementing both traits, and any type written after `largest` itself, all without recompiling or re-checking the function's body again for each.

10.5 Worked Example: A Bound That Grows With What the Body Needs

```
fn largest_display<T: PartialOrd + Copy + Display>(items: &[T]) -> String
    {
    let mut max = items[0];
    for &item in items.iter() {
```

```
        if item > max {
            max = item;
        }
    }
    format!("largest: {max}")
}
```

Adding the call to `format!("max")` requires the ability to display `T`, an operation not guaranteed by `PartialOrd` or `Copy` alone. The compiler will refuse to compile this function without also adding a `Display` bound; there is no way to accidentally use an operation the bound does not license, since Section 10.3's guarantee, that the bound alone determines everything the body can do, is enforced at every single use inside the function body, not merely at the boundary.

10.6 Interpretation Exercise: A Bound Broader Than the Body Needs

```
fn first<T: PartialOrd + Copy + Display + Clone + Default>(items: &[T]) ->
    T {
    items[0]
}
```

This compiles. The question worth asking is: *which of these bounds is Section 10.3's parametric admissibility argument actually using?* The function body only reads `items[0]` and returns it; nothing in the body compares, displays, clones, or default-constructs anything. Every bound beyond a basic copy-or-move requirement is unused by the implementation, yet each one narrows $\mathcal{U}$, the set of types the function can be called with, excluding any type that fails to implement even one of the unused traits. An over-broad bound does not make the function incorrect, but it silently shrinks the very set of types Section 10.2 argued was the whole benefit of writing generically in the first place; a caller with a type that satisfies everything `first` actually needs, but not `Display` or `Default`, is turned away for no reason the function's own logic can justify.

A Further Worked Example: The Iterator Trait as Parametric Admissibility

The standard library's `Iterator` trait is, in this chapter's vocabulary, close to the purest instance of parametric admissibility Rust ships with. The trait requires exactly one method, `fn next(&mut self) -> Option<Self::Item>`, and yet any type implementing that single method automatically gains dozens of further methods, `map`, `filter`, `fold`, `take`, `zip`, and more, none of which the implementer writes. This is only possible because every one of those additional methods is itself generic, verified once against the single required method's contract, exactly Section 10.3's $\forall\, S \in \mathcal{U}$ claim: the proof that `map` behaves correctly was carried out once, for any type satisfying `Iterator`'s one-method bound, and it applies to every iterator anyone has written or will ever write, without the standard library's authors needing to know in advance what those types would be.

```
struct Countdown(u32);

impl Iterator for Countdown {
    type Item = u32;
```

```
    fn next(&mut self) -> Option<u32> {
        if self.0 == 0 { None } else { self.0 -= 1; Some(self.0 + 1) }
    }
}

let total: u32 = Countdown(5).map(|n| n * 2).sum();
// map and sum were never written for Countdown specifically;
// both are verified once, generically, against Iterator's contract.
```

An informal law is worth stating explicitly here, since `map` is usually learned by memorizing examples rather than by noticing what makes it trustworthy. If a container preserves the structure of what it holds while letting its contents be transformed, then composing two transformations before mapping should produce the same result as mapping one transformation, then mapping the other over what results. Formally, for a container C and transformations f and g, mapping $g \circ f$ over C should equal mapping f over C and then mapping g over the result. `Iterator`'s `map`, and every other type this book has shown implementing something `map`-shaped, `Option`, `Result`, satisfies this law as a matter of course, not as an obligation the programmer separately has to prove; it falls out directly from `map` applying its function once per element, or once per contained value, with nothing else going on that could make the two orderings differ.

Connection to Category Theory

A structure satisfying the law just stated is called a *functor*. Rust's `Iterator`, `Option`, and `Result` all qualify, and this is precisely the property Section 10.3's parametric-admissibility argument was quietly relying on: a bound proven once over `map`'s behavior holds for every functor-shaped type that satisfies it, known or not. Bartosz Milewski's *Category Theory for Programmers* develops functors, and the richer structures built on top of them, in full generality [9].

Recurring Example: The Library System

The catalog search that closes this recurring example ties Chapters 6 through 10 together in one function. A single search routine should work across books, DVDs, and journals alike, bounded only by Chapter 9's `Searchable` contract, in exactly Section 10.3's parametric-admissibility sense:

```
trait Searchable {
    fn matches(&self, query: &str) -> bool;
}

fn find_all<T: Searchable>(catalog: &[T], query: &str) -> Vec<&T> {
    catalog.iter().filter(|item| item.matches(query)).collect()
}
```

`find_all` was verified once, against `Searchable`'s single required method, and that one verification already covers every type the library's catalog will ever hold, including a media format not yet acquired when this function was written, exactly Chapter 10's claim that a bound proven once holds for every type that satisfies it, known or not. This is where the recurring example set out in Chapter 1 arrives: a checkout desk's software, built entirely from the

shapes and contracts Chapters 6 through 10 introduced, structs for identity, enums for status, matching for reconstruction, traits for shared behavior, generics for code that works across every item type the catalog will ever hold, with none of Chapter 3's stale-belief races and none of Chapter 4's unbounded reasoning footprints, because Chapter 5's ownership discipline was there underneath all of it from the start.

10.7 Connections: Generics as Locality Extended Over a Family of Types

Chapter 9 bounded a function's reasoning footprint to a single trait's contract. This chapter has shown the same bound extended across an entire family of types at once, verified a single time rather than once per type. The cost of an overly broad bound, shown in Section 10.6, is a direct type-level analogue of Chapter 4's locality failures: just as a function that silently depends on more program state than its signature admits becomes harder to reason about as the program grows, a generic function that demands more from its type parameter than its body actually uses becomes harder to reuse as the ecosystem of types around it grows. Chapter 11 turns to composition, asking how several narrow trait bounds combined together tend to preserve this locality better than a single broad interface inherited wholesale.

Invariant Established in This Chapter

A constraint stated once over a type parameter holds for every type that satisfies it, known or not yet written. A generic function's admissible operations are exactly what its trait bounds supply, verified once against the bound rather than separately against each instantiating type; the bound should be exactly as wide as the body's actual operations require, since any bound wider than that excludes otherwise-eligible types for no reason the implementation itself can justify.

Chapter 11: Composition Over Inheritance

From First Principles

Question: If type B inherits from type A, does B gain A's capabilities, or does B inherit A's obligations along with them?

11.1 Phenomenon: The Base Class That Cannot Be Narrowed

A large kitchen appliance company sells a single "Base Blender" whose specification lists thirty methods: blend, puree, crush ice, knead dough, heat, cool, keep warm, and so on. Every appliance the company later releases is defined as a specialization of the Base Blender, inheriting all thirty methods whether or not the new appliance can sensibly perform them. When the company introduces a simple hand-held milk frother, it must still, by the terms of the inheritance hierarchy, answer to "knead dough" and "crush ice," either by implementing nonsensical behavior for operations the frother cannot perform, or by throwing an error at the moment someone tries. Either choice is a symptom of the same underlying problem: the frother did not choose to take on thirty obligations, it inherited them, because inheritance in this system offers no way to take only what is needed.

A zoo's classification of "flightless bird" inheriting from "bird," which itself declares a `fly` method, runs into the identical wall: an ostrich is, by any reasonable definition, a bird, but the inheritance hierarchy forces every bird to answer to `fly`, leaving the ostrich's implementation to either do something incoherent or fail loudly. The problem in both cases is not that the specific hierarchy was poorly designed; it is that single-parent inheritance bundles an entire package of capabilities as an indivisible unit, and any type that wants *some* of that package is forced to inherit *all* of it.

Chapter 9 introduced traits as narrow, independently statable contracts. This chapter asks what happens when several such contracts are combined for a single type, rather than inherited wholesale from one broad ancestor, and argues that the difference is not stylistic but structural: composition lets a type select exactly the obligations it can honor, while inheritance forces it to accept a fixed bundle chosen in advance by someone who could not have anticipated every future specialization.

11.2 General Principle: Obligations Should Be Selected, Not Inherited Wholesale

Chapter 9 established that a trait's contract restricts a caller's admissible continuations to exactly the trait's declared methods. A type that implements several small, independent traits is making several small, independent promises, each of which can be verified, understood, and relied upon in isolation. A type built by inheriting from one large base type is instead making one large, bundled promise, and the bundling itself is where the frother-and-blender problem originates: nothing about the bundle's internal structure lets a subtype accept part of it while declining the rest.

The deeper issue is one of distinguishability. A base class's thirty methods are not, in general, thirty independent facts about the world; some belong together because they genuinely co-occur in every sensible specialization, and others were bundled together only because the original author happened to place them on the same class. Composition, by contrast, forces the author to state each capability as its own trait, which means the boundaries between capabilities have to be decided deliberately, one at a time, rather than inherited as an undifferentiated mass from whatever the base class happened to contain.

11.3 Formal Definition: Composed Versus Inherited Capability Sets

Let a type S's full capability set be the set of all trait contracts it honors, $\text{Cap}(S) = \{\mathcal{T}_1, \mathcal{T}_2, \ldots\}$. Under composition, $\text{Cap}(S)$ is built by S's author selecting each $\mathcal{T}_i$ independently, and each is verified independently against S's implementation, in the sense of Chapter 9. Under single-inheritance from a base type B with a fixed bundle $\text{Cap}(B) = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$, every subtype S of B is constrained to satisfy $\text{Cap}(B) \subseteq \text{Cap}(S)$ as an indivisible unit; there is no mechanism by which S can inherit $\mathcal{T}_1$ and $\mathcal{T}_3$ from B while declining $\mathcal{T}_2$, even if $\mathcal{T}_2$ is meaningless for S.

This is the formal shape of the frother problem: the frother's ideal capability set is some proper subset of the blender's, but single inheritance offers only the full bundle or nothing, since the bundle was fixed once, at the base type's definition, rather than assembled independently at each subtype's definition the way a composed capability set is.

11.4 Rust Mechanism: Composing Independent Trait Bounds

```
trait Blend {
    fn blend(&self);
}

trait CrushIce {
    fn crush_ice(&self);
}

trait KneadDough {
    fn knead_dough(&self);
}

struct MilkFrother;
```

```
impl Blend for MilkFrother {
    fn blend(&self) {
        // froths milk using the same underlying motor action
    }
}
// MilkFrother implements only Blend. There is no mechanism, and no
// obligation, requiring it to also implement CrushIce or KneadDough;
// each trait was an independent choice, made once, by MilkFrother's
// own author.
```

Rust has no single-inheritance mechanism for structs at all; every capability a type has is the result of an explicit `impl` block for some trait, chosen independently. The frother-and-blender problem from Section 11.1 cannot arise in this style of composition, not because Rust's authors were especially disciplined, but because the language provides no bundling mechanism through which it could arise in the first place.

11.5 Worked Example: Combining Capabilities Without a Shared Ancestor

```
struct StandMixer;

impl Blend for StandMixer {
    fn blend(&self) {}
}
impl CrushIce for StandMixer {
    fn crush_ice(&self) {}
}
impl KneadDough for StandMixer {
    fn knead_dough(&self) {}
}

fn run_full_cycle(m: &(impl Blend + CrushIce + KneadDough)) {
    m.blend();
    m.crush_ice();
    m.knead_dough();
}
```

`run_full_cycle` requires all three capabilities simultaneously, expressed as a trait bound combining three independent contracts with `+`, exactly as Chapter 10 combined `PartialOrd` and `Copy`. `StandMixer` satisfies this bound because it happens to implement all three traits, not because it inherited from some common ancestor that bundled them together; a hypothetical fourth type could satisfy the same bound by implementing the same three traits through an entirely unrelated internal representation, since, as Chapter 9 established, the bound cares only about the contract, never about shared ancestry.

11.6 Interpretation Exercise: A Composition That Reveals a Missing Trait

```
struct ImmersionBlender;

impl Blend for ImmersionBlender {
    fn blend(&self) {}
}
impl CrushIce for ImmersionBlender {
    fn crush_ice(&self) {}
}

fn run_full_cycle_2(m: &(impl Blend + CrushIce + KneadDough)) {
    m.blend();
    m.crush_ice();
    m.knead_dough();
}

// run_full_cycle_2(&ImmersionBlender); // does not compile
```

The commented-out call fails to compile. The question worth asking is not simply why, since the reason is visible in the bound, but: *what has the compiler's rejection actually told the programmer that a base-class hierarchy would have hidden?* Under single inheritance, if `ImmersionBlender` had been forced to inherit from a base `Appliance` type bundling all three operations, the missing `knead_dough` capability would either have compiled with a nonsensical implementation or failed at runtime the first time it was invoked. Under composition, the absence is caught at the exact call site that required it, phrased in terms of exactly the capability that is missing, `KneadDough`, rather than as a vague failure somewhere inside an inherited method nobody meant to call. The rejection is not a limitation of composition; it is composition correctly reporting that `ImmersionBlender`'s true capability set, honestly stated, never included dough-kneading in the first place.

11.7 Connections: Composition as Distinguishability Preserved

This chapter closes Part III by tying together Chapters 6 through 10. Structs and enums, from Chapters 6 and 7, declare a value's admissible *data* shape without conflating unrelated fields or variants. Traits, from Chapter 9, declare admissible *behavior* without conflating unrelated capabilities. Generics, from Chapter 10, extend a single trait bound across an open-ended family of types. Composition, this chapter's subject, is what happens when several such behavioral contracts are combined for one type: each remains independently named, independently verified, and independently omittable, which is exactly what allows a type's true capability set to be stated honestly rather than approximated by the nearest available bundle. Part IV turns from composing capabilities to a related question: what happens when an operation's own result is not guaranteed to succeed, and how that partiality should be represented rather than hidden.

Invariant Established in This Chapter

A type's capabilities should be composed from independently stated, independently verified contracts, not inherited as a fixed bundle chosen in advance by an ancestor that could not anticipate every future specialization. Composition lets a type's capability set be exactly what its author can honestly implement; single-inheritance bundling forces a choice between accepting obligations the type cannot meaningfully satisfy or fracturing the hierarchy to avoid them, and the resulting gap is often discovered at runtime rather than, as with composition, at the precise call site that required the missing capability.

Part III Summary

- A struct's state space is exactly the product of its declared fields, no more and no less (Chapter 6).
- An enum's state space is a closed sum of named alternatives, and every continuation must be explicit (Chapter 7).
- Pattern matching reconstructs which alternative a value occupies, and is only trustworthy when exhaustive (Chapter 8).
- A trait restricts a caller's admissible continuations to exactly its declared contract (Chapter 9).
- A bound proven once over a type parameter holds for every type that satisfies it, known or not (Chapter 10).
- Capabilities should be composed from independent contracts, not inherited as a fixed bundle (Chapter 11).

Therefore: a type system that wants composed, verifiable behavior must let both data and behavior be declared narrowly and combined explicitly, rather than assumed complete or inherited wholesale from an ancestor that could not anticipate every future specialization. Part IV turns from what a value *is* to what happens when a function cannot always produce one.

Part IV

Programs as Constraint Systems

Chapter 12: Partial Functions and the Edge of Definition

From First Principles

Question: If a function's signature says it accepts any value of a type, but some of those values make it fail, was the signature actually telling the truth?

12.1 Phenomenon: Domains That Are Narrower Than They Look

A vending machine's coin slot accepts anything shaped like a coin, but it does not accept every such object equally: a foreign coin of the wrong diameter, a washer, a warped or bent coin, all fit the slot's opening yet fail somewhere downstream, rejected by a mechanism the slot's shape gave no hint of. The slot's physical shape describes an outer boundary, everything that can be inserted, but not the true boundary of what the machine can actually process; that narrower boundary is discovered only by the coin's journey through the mechanism, not by inspecting the slot.

A recipe that calls for "juice of one lemon" presupposes, silently, that the lemon has juice to give: a lemon that has already been juiced, or one that has dried out on the counter for a month, fits the recipe's stated input, a lemon, while failing to satisfy what the instruction actually needs. The recipe's stated domain, any lemon, is broader than its true domain, a lemon capable of yielding juice, and the gap between the two is exactly where an inexperienced cook is caught off guard.

Chapter 1 defined a program as a function $P : \Sigma \to \Sigma$, and noted in passing that such functions may be *partial*: defined for some states and not others. This chapter examines that partiality directly, because most of the software defects that make it to production are not failures of logic within a function's defined domain, they are failures at exactly the boundary this section's examples describe: a caller supplying an input that type-checks, that fits the declared shape, but that lies outside the function's true, narrower domain of correct behavior.

12.2 General Principle: Every Function Has an Honest Domain and a Claimed One

Call a function's *claimed domain* the set of inputs its type signature admits, and its *honest domain* the set of inputs for which it actually produces a meaningful result without crashing,

looping forever, or silently returning nonsense. For a great many functions in a great many languages, the honest domain is a strict subset of the claimed domain: division admits any pair of numbers by its signature but is undefined when the divisor is zero; indexing into a collection admits any integer by its signature but is undefined once the index exceeds the collection's length; parsing a string into a number admits any string by its signature but is undefined for a string that is not, in fact, a well-formed number.

The gap between the claimed domain and the honest domain is not a minor implementation detail. It is the single most common source of the runtime failures Chapter 4 warned about when it described action at a distance: a function whose claimed domain silently overstates its honest domain is a function whose type signature is, in a precise sense, lying to every caller who trusts it, and the caller has no way to discover the lie except by encountering an input that exposes it, typically much later, and typically not at the call site where the mismatched input actually originated.

12.3 Formal Definition: Partiality as a Domain Restriction

A function $f : A \to B$ is *total* if it is defined for every $a \in A$, and *partial* if there exists some $a \in A$ for which $f(a)$ is undefined, meaning f either fails to terminate, terminates by crashing the program, or, worst of all, terminates with a result that is silently wrong. Write $\text{dom}(f) \subseteq A$ for f's honest domain, the subset on which f genuinely behaves as its documentation or its author's intent describes. A signature $f : A \to B$ is fully honest exactly when $\text{dom}(f) = A$; it is dishonest, in the sense this chapter cares about, whenever $\text{dom}(f) \subsetneq A$ and nothing in the type of f reflects that restriction.

This gives a precise target for what a language's type system could, in principle, do about partiality: either shrink A itself, by encoding the restriction into the input type so that only genuinely valid inputs type-check at all, or widen B, the output type, to include an explicit representation of the cases where f cannot produce a meaningful result, so that $\text{dom}(f) = A$ becomes true again, honestly, because failure has been folded into the codomain rather than left as an undocumented gap in the domain. The next several chapters, starting with `Option` in Chapter 13 and `Result` in Chapter 14, are two different instances of this second strategy.

12.4 Rust Mechanism: A Type Signature That Still Lies

```
fn first_char(s: &str) -> char {
    s.chars().next().unwrap()
    // Claimed domain: any &str.
    // Honest domain: any non-empty &str.
    // Called with "", this panics: the type signature promised
    // a char for every string, and an empty string exposes that
    // the promise was never actually kept.
}
```

Rust's type system does not, by itself, prevent this kind of dishonest signature; `fn first_char(s: &str) -> char` type-checks perfectly well despite its honest domain being strictly narrower than its claimed one. What Rust does provide, and what the remainder of Part IV develops,

is a standard, idiomatic vocabulary for closing this gap deliberately: `Option<char>` as a return type would have made the restriction visible in the signature itself, converting a hidden partiality into a declared one.

12.5 Worked Example: Narrowing the Input Instead of Widening the Output

Section 12.3 named two strategies: shrink A, or widen B. Widening B is the more common idiom in Rust and is covered starting in Chapter 13; shrinking A is less common but instructive, because it moves the honesty even further upstream, to the moment of construction rather than the moment of use.

```
struct NonEmptyStr<'a>(&'a str);

impl<'a> NonEmptyStr<'a> {
    fn new(s: &'a str) -> Option<Self> {
        if s.is_empty() { None } else { Some(NonEmptyStr(s)) }
    }
}

fn first_char_total(s: NonEmptyStr) -> char {
    s.0.chars().next().unwrap()
    // Claimed domain: any NonEmptyStr.
    // Honest domain: the same set, exactly, because NonEmptyStr's
    // own constructor already excluded the one input that used
    // to cause a panic. The .unwrap() here cannot fail, and a
    // reader can verify that by reading NonEmptyStr::new alone,
    // without needing to trust first_char_total's implementation.
}
```

This is Chapter 6's guarded-construction pattern, from Section 6.5, applied directly to the partiality problem: by narrowing A at the type level so that only genuinely valid inputs can be constructed at all, `first_char_total` becomes honestly total over its new, narrower claimed domain, and the internal `.unwrap()` stops being a hidden risk and becomes a locally verifiable fact.

Historical Note: Ariane 5 Flight 501

In 1996, the maiden flight of the Ariane 5 rocket self-destructed roughly thirty-seven seconds after launch. The cause traced back to a single conversion: a 64-bit floating-point value representing horizontal velocity was converted to a 16-bit signed integer, using code inherited, unexamined, from the earlier Ariane 4, whose flight profile never produced a velocity value large enough to overflow that conversion. Ariane 5's flight profile did. The conversion function's claimed domain, in this chapter's vocabulary, was "any `f64`"; its honest domain was only values representable in a 16-bit signed integer, and the gap between the two was never represented in the function's own signature, so nothing forced anyone re-examining the inherited code to notice it had changed.

Rust's ordinary `as` numeric cast has the same shape of danger the Ariane 5 conversion did: `value as i16` silently truncates or saturates an out-of-range `f64` rather than reporting

a problem, exactly the dishonest-signature pattern this chapter opened with. The idiomatic alternative makes the honest domain visible in the type itself:

```
use std::convert::TryFrom;

fn convert_velocity(v: f64) -> Result<i16, std::num::TryFromIntError> {
    i16::try_from(v as i64)
}
```

`i16::try_from` returns a `Result`, in exactly the sense Chapter 14 develops at length, forcing the out-of-range case to be handled at the call site rather than silently truncated and passed on as though it were valid. The Ariane 5 failure was not a mistake in the arithmetic; it was a signature, inherited without review, that claimed a domain far wider than the one it could honestly support.

12.6 Interpretation Exercise: A Signature That Looks Narrowed but Isn't

```
struct NonEmptyStr<'a>(pub &'a str);
```

Suppose `NonEmptyStr`'s field were made `pub`, as in this line. The question worth asking is: *does `first_char_total` remain honestly total after this change?* It does not, and the reason is the same one Section 6.6 already worked through for a different type: a public field reopens direct construction, `NonEmptyStr("")`, entirely bypassing `new` and its emptiness check. Section 12.3's honest-domain guarantee for `first_char_total` was never actually a property of the function itself; it was borrowed, entirely, from the guarantee that every `NonEmptyStr` in existence had passed through `new`. The moment that guarantee is reopened at the field level, the partiality this chapter is concerned with does not reappear inside `first_char_total`'s own logic, it reappears exactly where Chapter 6 warned it would: at the boundary of the type whose job was to keep it closed.

12.7 Connections: Opening Part IV

This chapter has deliberately stopped short of introducing `Option` and `Result` as language features; it has instead built the vocabulary, claimed domain versus honest domain, that makes clear *why* they exist and what problem they are actually solving. Chapter 13 takes up `Option` as a way of widening a function's codomain to honestly represent the possibility of absence, reading it as an application of Chapter 7's enum machinery to exactly the domain-narrowing gap this chapter has named. Chapter 14 takes up `Result` the same way, for the case where failure carries information about what went wrong rather than merely that something did.

Invariant Established in This Chapter

Every function has an honest domain, the inputs for which it actually behaves as claimed, and this is frequently narrower than the claimed domain its type signature admits. A signature is dishonest whenever this gap exists and is left unrepresented in the types; closing the gap requires either narrowing the input type so only genuinely valid values can be constructed, or widening the output type to represent failure explicitly, and the chapters that follow develop the second strategy as Rust's primary idiom.

Chapter 13: Option as Topological Incompleteness

From First Principles

Question: Is the absence of a value a special case of having a value, or is it a genuinely different kind of thing that has been forced to share a type with values it is not?

13.1 Phenomenon: A Hole Mistaken for a Point

A parking lot with two hundred numbered spaces and a sign reading "current occupant of space 57" presupposes that space 57 has an occupant to name. On any afternoon when space 57 happens to be empty, the sign's implicit type, a car, has no value to display, and whatever convention the lot uses to handle this, an empty placeholder sticker, a blank space, a special "VACANT" plate, is doing real work that is easy to overlook precisely because it so rarely fails: the overwhelming majority of the time, space 57 does have an occupant, and the special case is forgotten until the one day it matters.

A census form's field for "name of spouse" presupposes a spouse exists. For an unmarried respondent, the field has nothing correct to contain, and a form that provides no way to indicate this, forcing every respondent to write *something*, will accumulate garbage data: blank strings that might mean unmarried, might mean the respondent skipped the question, or might mean data entry lost the answer. The field's type, implicitly "a name," was never actually broad enough to represent every respondent honestly, and the gap was filled, badly, by convention rather than by design.

The same hole reappears anywhere an object's presence is the thing actually in question. A household mailbox is either holding mail or it isn't, and a mail carrier's route depends on being able to tell the two apart rather than treating an empty mailbox as a malfunctioning one. A theater seat is either occupied or vacant, and a box office's seating chart needs a real, first-class representation of "vacant," not a seat record with a blank or placeholder name in it, or the chart cannot be trusted to tell an usher where to actually seat the next patron.

Chapter 12 named this general shape as the gap between a claimed domain and an honest domain, but absence deserves its own chapter because it is, by a wide margin, the single most common instance of the problem, and because many languages provide a uniquely bad way of handling it: a null value that inhabits every reference type simultaneously, indistinguishable at the type level from a value that is actually present, discovered only when code that assumed presence is handed absence instead and fails, often far from where the absent value first arose.

13.2 General Principle: A Hole in a State Space Should Be Named, Not Hidden

Chapter 6 defined a struct's state space as everything the product of its fields admits, and Chapter 7 defined an enum's state space as the disjoint sum of its variants; in both cases, the state space was exactly what the type declared, no more and no less. A type T that sometimes has "no value of T" as a legitimate outcome is, considered honestly, not fully described by T alone; there is a hole in T's state space where a value was expected but none exists, and Section 13.1's examples showed what happens when that hole is patched informally, with a sentinel value, a null reference, or a silently blank field, rather than named as a first-class member of the type.

The alternative is to stop treating T as the complete type and instead declare a new type whose state space is T's state space with the hole made explicit: every legitimate value of T, plus one additional, clearly distinguished state meaning "no value here." This is not a workaround bolted onto T; it is the honest state space T should have had from the start, wherever absence was a real possibility, made visible rather than smuggled in through convention.

13.3 Formal Definition: Option as a Space With a Distinguished Hole

For a type T with state space Σ_T, define

$$\Sigma_{\texttt{Option<T>}} = \Sigma_T \sqcup \{\, \texttt{None} \,\},$$

a disjoint sum, in exactly the sense Chapter 7 defined for enums generally, between T's ordinary values and a single new, distinguished element representing absence. Borrowing the topological language this chapter's title invokes: if Σ_T is thought of as a space of admissible values, a partial function that sometimes has nothing meaningful to return has, in effect, a hole in that space, a location its honest domain does not cover. `Option<T>` does not fill the hole with a fabricated value of T; it names the hole as its own distinct point, `None`, external to Σ_T but internal to the larger, honestly complete space $\Sigma_{\texttt{Option<T>}}$.

This is why `Option<T>` is a strictly different type from T, rather than T with a special case, and why Rust does not allow a bare T to silently stand in for `Option<T>` or vice versa: doing so would be exactly the conflation Section 13.1's parking lot and census form examples illustrated, treating a hole as though it were an ordinary point of the space.

13.4 Rust Mechanism: The Compiler Refuses to Assume the Hole Is Filled

```
fn find_spouse(records: &[(String, Option<String>)], name: &str) -> Option
    <String> {
    records.iter()
        .find(|(n, _)| n == name)
        .and_then(|(_, spouse)| spouse.clone())
}
```

The return type `Option<String>` states, honestly, that this function's result may be a name or may be nothing at all; there is no way to call code that receives this result as though it were a plain `String` without first passing through some form of the case analysis Chapter 8 described, typically a `match` or one of `Option`'s combinator methods. The compiler will not compile code that reaches for a spouse's name without first confirming, in a way it can verify, that the `None` case has been considered.

13.5 Worked Example: Combinators as Case Analysis in Disguise

```
fn greeting(records: &[(String, Option<String>)], name: &str) -> String {
    match find_spouse(records, name) {
        Some(spouse) => format!("Hello, {name} and {spouse}!"),
        None => format!("Hello, {name}!"),
    }
}
```

This `match` is a direct instance of Chapter 8's reconstruction pattern applied to the two-variant sum from Section 13.3: the `Some` arm reconstructs the fact that a spouse's name is genuinely present, licensing the binding `spouse: String` without further checking, exactly as a `Circle` arm in Chapter 8 licensed a `radius` binding. The `None` arm is not an afterthought bolted on to satisfy the compiler; it is the case analysis's other, equally real branch, corresponding to the hole Section 13.3 named explicitly.

13.6 Interpretation Exercise: A Combinator That Quietly Refills the Hole

```
fn greeting_v2(records: &[(String, Option<String>)], name: &str) -> String
    {
    let spouse = find_spouse(records, name).unwrap_or_default();
    format!("Hello, {name} and {spouse}!")
}
```

This compiles and, for an unmarried respondent, produces something like "Hello, Alex and !", silently. The question worth asking is: *has this code actually handled the `None` case, or has it merely made the hole invisible again?* `unwrap_or_default` does perform an exhaustive case analysis in the formal sense, both variants of `Option<String>` are accounted for, so Chapter 8's exhaustiveness guarantee is technically satisfied. But the value substituted for `None`, an empty string, is not a meaningful member of the honest domain the greeting was trying to express; it is a fabricated point standing in for the hole, precisely the move Section 13.2 argued against, now performed one layer up, inside a combinator that looks like careful handling but has, in substance, recreated the parking lot's blank sticker. The lesson is that exhaustiveness alone is necessary but not sufficient; each arm's content, not merely its existence, has to honestly represent what that branch of the case analysis means.

Connection to Category Theory

`Option<T>`, together with the chain of `Some`/`None`-aware combinators built on top of it, is category theory's *Maybe* monad: a functor, in Chapter 10's sense, that additionally supports sequencing operations which may themselves fail to produce a value. `None`, formalized in Section 13.3 as a distinguished point in a widened codomain, is exactly `Maybe`'s `Nothing` constructor [9].

13.7 Connections: The First of Two Widened Codomains

Chapter 12 named two strategies for closing the gap between a claimed domain and an honest one: narrowing the input type, which Chapter 12 illustrated directly, or widening the output type, which this chapter has now shown in its simplest form. `Option<T>` widens a codomain just enough to represent one specific kind of failure, the absence of any meaningful result, without saying anything about *why* the result is absent. Chapter 14 takes up `Result`, which widens a codomain further still, attaching a reason to the failure case rather than collapsing every kind of failure into a single, undifferentiated hole.

Erik Meijer has argued, from the vantage point of a language designer who has worked closely with both pure and impure styles, that most real programs are honestly a mix of total and partial computation, and that a language serves its users badly if it forces a false choice between pretending every function is total and giving up on tracking partiality altogether [10]. `Option<T>` is Rust's answer to that tension in miniature: it does not pretend absence cannot happen, in the way a null-permitting language's ordinary reference types implicitly do, and it does not force every function to be rewritten as an exception-throwing escape from the type system either. It gives partiality a type, and lets a function be honestly partial without being informally, silently partial.

Invariant Established in This Chapter

A type's state space should honestly represent every outcome a function can produce, including the absence of a meaningful value; where such a hole exists, it should be named as a distinct, compiler-checked member of a widened type rather than patched informally with a sentinel value, a null reference, or a silently fabricated default. `Option<T>` is the disjoint sum of T's ordinary values with a single distinguished point, `None`, and handling it honestly requires that every arm of the resulting case analysis, not merely its presence, actually represent what that branch means.

Chapter 14: Result as Boundary Propagation

From First Principles

Question: When something goes wrong at the edge of a system, whose job is it to know why?

14.1 Phenomenon: Failures That Arrive Without a Cause

A cargo manifest that only ever records "delivered" or "not delivered" tells a warehouse manager that a shipment failed to arrive, but nothing about whether it was lost, stolen, damaged in transit, refused at customs, or simply delayed. Each of these causes calls for a different response: a damaged shipment needs an insurance claim, a customs refusal needs different paperwork, a delay needs only patience. A manifest that collapses all of them into a single "not delivered" status has technically reported the failure, in the sense that nobody is left thinking the shipment arrived, but it has stripped out exactly the information needed to act on the failure correctly.

A smoke detector that only ever beeps, with the same tone whether the battery is low, the sensor has failed, or there is an actual fire, has represented the presence of a problem while discarding the one piece of information, which problem, that determines what the occupant should do next. A passport application that is simply rejected, with no accompanying reason, whether an expired supporting document, a photo that failed the size requirement, an unpaid fee, leaves the applicant unable to correct anything before resubmitting. A bank transfer that fails and reports only "failed" rather than distinguishing insufficient funds from an invalid account number from a frozen account leaves the sender guessing at a problem the bank's own system already knew precisely. Chapter 13 addressed the case where a function has nothing at all to report, absence. This chapter addresses the more common and more consequential case: a function has something to report, a genuine failure, but that failure carries a reason, and collapsing the reason away, the way `Option`'s bare `None` would, throws away exactly the information a caller needs to respond appropriately rather than merely to notice that something went wrong.

What Goes Wrong Without This: Ignored Failures

Before this chapter's mechanism is introduced, it is worth seeing the failure mode that motivates it directly: a language where a fallible operation's failure can simply be left unchecked, silently, with the program proceeding as though it had succeeded.

```
// Hypothetical, not valid Rust:
// let file = open_file("config.json"); // may fail; return value ignored
// let contents = file.read_all();       // reading a possibly-invalid
   handle
// let config = parse(contents);         // parsing possibly-garbage data
// -- if open_file failed, every line after it is operating on nonsense,
// -- and nothing forced anyone to notice at the point where the
// -- failure actually happened
```

Nothing about this code is obviously wrong to read; it looks like an ordinary, linear sequence of steps. The danger is precisely that the failure at the first line is not required to be examined before the program continues, so an error that occurred at `open_file` does not surface until `parse` produces a confusing, unrelated error several steps later, or, worse, silently produces a wrong but plausible-looking result. Chapter 14's `Result` type exists specifically to close this gap: a value of type `Result<T, E>` cannot be used as a `T` without the program first accounting, in code the compiler checks, for the possibility that it is actually an `E`.

14.2 General Principle: A Boundary Should Propagate What It Knows

Call any point in a program where control passes from one component to another, a function call, a network request, a file read, a *boundary*. Chapter 2 argued that composing components reliably requires each one's declared state to actually capture what it depends on and produces; this chapter extends that argument to failure specifically. When a component fails at a boundary, it typically knows something about why: a file boundary knows whether the failure was "not found" or "permission denied"; a network boundary knows whether the failure was a timeout or a malformed response. This knowledge is cheapest to capture at the exact boundary where the failure occurred, because that is where the underlying cause is directly observable; the moment the failure is reported upward as an undifferentiated signal, that knowledge is gone, and no amount of code downstream can reconstruct it.

The obligation this creates is symmetric to the one Chapter 3 described for mutation: just as a mutation should not silently invalidate an observer's beliefs, a failure should not silently discard information the boundary already possessed. A boundary that swallows the cause of its own failure is imposing exactly the kind of unbounded, undocumented cost on its callers that Chapter 4 spent an entire chapter arguing against: every caller now has to guess at, or separately re-derive, information the boundary already had in hand and simply declined to pass along.

14.3 Formal Definition: Result as a Sum That Carries Evidence

For a success type T and an error type E, define

$$\Sigma_{\texttt{Result<T,E>}} = \Sigma_T \sqcup \Sigma_E,$$

a disjoint sum in the same sense as Chapter 13's `Option<T>`, but with the second summand no longer a single distinguished point like `None`; it is an entire type E, capable of carrying whatever evidence about the failure the boundary that produced it chose to attach. Where `Option<T>` widened a codomain just enough to name a hole, `Result<T, E>` widens it enough to name the hole *and* attach a reconstructable account of why the hole is there.

This distinction matters for exactly the reason Section 14.1's cargo manifest illustrated: a function returning `Option<T>` has told its caller only that $T \notin \mathrm{dom}(f)$ for this input, in Chapter 12's vocabulary, while a function returning `Result<T, E>` has told its caller which of potentially many reasons explains why. The choice between the two types is, in effect, a decision about how much of the boundary's own knowledge is worth propagating, and Section 14.6 examines what is lost when that decision is made carelessly.

14.4 Rust Mechanism: The ? Operator as Propagation, Not Absorption

```
use std::fs;
use std::io;

fn read_first_line(path: &str) -> Result<String, io::Error> {
    let contents = fs::read_to_string(path)?;
    Ok(contents.lines().next().unwrap_or("").to_string())
}
```

The `?` operator after `fs::read_to_string(path)` does not discard a failure and substitute some generic error; if the read fails, the `?` operator returns immediately from `read_first_line`, carrying the original `io::Error`, with its specific cause (not found, permission denied, and so on) intact, out to `read_first_line`'s own caller. This is Section 14.2's boundary-propagation principle enforced directly by the language: the evidence the file-system boundary produced is neither inspected nor discarded by this function, only forwarded, exactly preserved, to whichever caller is actually in a position to decide what to do about it.

14.5 Worked Example: Preserving Evidence Across Multiple Boundaries

```
use std::fmt;

#[derive(Debug)]
enum ConfigError {
    Io(io::Error),
    Parse(std::num::ParseIntError),
}

impl fmt::Display for ConfigError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            ConfigError::Io(e) => write!(f, "could not read config: {e}"),
```

```
            ConfigError::Parse(e) => write!(f, "invalid config value: {e}"
                ),
        }
    }
}

fn read_max_connections(path: &str) -> Result<u32, ConfigError> {
    let contents = fs::read_to_string(path).map_err(ConfigError::Io)?;
    contents.trim().parse::<u32>().map_err(ConfigError::Parse)
}
```

Two different boundaries are crossed here, a file read and a string-to-integer parse, each with its own distinct failure mode. `ConfigError` is itself an enum, in Chapter 7's sense, whose two variants preserve the origin-specific evidence from each boundary rather than collapsing both into an undifferentiated "config error." A caller of `read_max_connections` that matches on the returned `ConfigError` can distinguish "the file was missing" from "the file existed but contained garbage," a distinction the underlying boundaries actually knew and that Section 14.2 argued should not be discarded on the way up.

14.6 Interpretation Exercise: A Propagation That Quietly Becomes an Absorption

```
fn read_max_connections_v2(path: &str) -> Result<u32, String> {
    let contents = fs::read_to_string(path).map_err(|_| "config error".
        to_string())?;
    contents.trim().parse::<u32>().map_err(|_| "config error".to_string())
}
```

This compiles and satisfies the same `Result`-returning shape as Section 14.5's version. The question worth asking is: *has this function actually propagated the boundary's knowledge, or has it discarded it while keeping the type that looks like propagation?* Both `map_err` closures replace the original error, an `io::Error` or a `ParseIntError` carrying specific, structured information, with the same literal string, `"config error"`, regardless of which boundary failed or why. The function's signature, `Result<u32, String>`, still declares a widened codomain in the sense Section 14.3 described, and the exhaustiveness machinery from Chapter 8 still applies to it correctly. But the evidence Section 14.2 argued should be propagated has been deliberately erased at the exact boundary that possessed it, leaving every caller unable to distinguish a missing file from a malformed one, even though the underlying `fs::read_to_string` and `.parse()` calls knew the difference perfectly well. A `Result` type is only as informative as the error type propagated through it; the type system enforces that failure is represented, but it cannot enforce that the representation is honest about what actually happened.

A Further Worked Example: Serialization as a Boundary Contract

Deserializing external data, a JSON document from a network response, a configuration file, user-supplied input, is a boundary crossing of exactly the kind Section 14.2 described in the abstract: the incoming bytes' claimed domain, in Chapter 12's sense, is "any sequence of bytes claiming to be JSON," while the honest domain of "a value that actually populates every field

a Rust struct requires, with every field's declared type satisfied," is very much narrower. The `serde` ecosystem's `Deserialize` trait signature reflects this directly: deserialization is not a function from bytes to a value, it is a function from bytes to a `Result<T, Error>`, because the gap between the two domains is exactly what Chapter 14's entire argument says must be represented rather than assumed away.

```
#[derive(serde::Deserialize)]
struct Config {
    max_connections: u32,
    host: String,
}

fn load_config(json: &str) -> Result<Config, serde_json::Error> {
    serde_json::from_str(json) // the boundary's own Result, propagated
        untouched
}
```

Note what `load_config` does *not* do: it does not catch `serde_json::Error` and collapse it into a bare string, the way Section 14.6's `read_max_connections_v2` collapsed two distinct causes into `"config error"`. The structured error `serde_json` produces, which field was missing, which type mismatched, at which line and column, is exactly the kind of boundary-specific evidence Section 14.2 argued should survive the crossing, and a caller that needs to report a useful message to whoever supplied the malformed configuration depends on that evidence having been preserved rather than discarded at the first opportunity.

Connection to Category Theory

`Result<T, E>` is category theory's *Either* monad, and the `?` operator's propagation, examined in Section 14.4, is Kleisli composition: chaining functions that may each fail, threading the failure through automatically rather than requiring it to be checked and re-raised by hand at every step [9].

14.7 Connections: Failure as a First-Class Citizen of the State Space

Chapters 12 through 14 together complete a single argument. Chapter 12 named the gap between a claimed domain and an honest one; Chapter 13 showed the simplest way to close that gap, widening the codomain to name an unstructured hole; this chapter has shown the general case, widening the codomain to carry a structured account of why the hole exists.

This chapter's argument has a considerably higher-stakes analogue outside software, worth naming once, explicitly, rather than only implying. Catherine Liu's critique of what she calls the professional-managerial class describes a pattern of institutional abstraction in which structural and economic consequences are treated as someone else's department, formally accounted for while the specific evidence, whose cost, borne by whom, that would let anyone downstream trace an outcome back to the decision that produced it is quietly discarded along the way [4]. Sarah Wynn-Williams's account of decision-making inside a large technology company describes a structurally similar failure at organizational scale: choices made early, and abstracted away from their eventual human consequences at the moment they were made, propagate outward the way an under-specified error type propagates in Section 14.6, arriving

at the people who actually bear the cost with the evidence of who decided what, and why, already stripped from it [5]. Neither book is about software, and this chapter's technical argument does not depend on either of them. But the shape of the failure, evidence discarded at precisely the boundary where it was cheapest to preserve and most needed downstream, is the same failure at every scale this book has examined, from a single Rust function's error type to an organization's account of its own decisions.

Chapter 15 turns to the alternative Rust reserves for cases where a program has entered a state its own type system claimed was impossible, `panic`, and examines why that response is deliberately different in kind from returning a `Result`, rather than merely a more severe version of the same mechanism.

Invariant Established in This Chapter

Failure is represented rather than hidden. A boundary that fails typically knows something about why, and that knowledge is cheapest to preserve at the exact point where the failure occurs; propagating it faithfully, through a widened codomain that carries structured evidence rather than an undifferentiated signal, lets a caller respond to the actual cause rather than merely learn that something, unspecified, went wrong. A `Result` type only delivers on this guarantee to the extent that its error type honestly preserves what the boundary knew; the type system enforces that failure is represented, not that the representation retains its meaning.

Chapter 15: Panic as Leaving the Admissible Manifold

From First Principles

Question: If a type system can prove a certain state is impossible, what should happen when that state occurs anyway?

15.1 Phenomenon: Alarms That Mean the Model Was Wrong

A structural engineer's load calculations establish, in advance, a range of stresses a bridge's beams are designed to bear. A strain gauge that reports a stress outside that range is not reporting a new, slightly unusual data point to be smoothed into the ongoing average; it is reporting that one of the calculation's foundational assumptions, about the load, the material, or the beam's condition, has been violated in the world in a way the model did not anticipate. The correct response is not to keep the bridge open while investigating; it is to stop traffic immediately, because every further computation the engineers might perform about the bridge's safety was built on an assumption that has just been shown to be false.

A chemical plant's pressure relief system is calibrated to a maximum safe pressure derived from the vessel's known material properties. A reading beyond that threshold does not indicate a slightly higher than usual, but still manageable, operating condition; it indicates that the vessel has entered a regime the engineers who designed the relief system explicitly excluded from consideration, because no plan exists for operating safely there. The relief valve does not attempt to compensate gracefully; it vents immediately, because continuing to run the process under an assumption that has already been falsified is more dangerous than an abrupt, visible stop.

Chapters 5 through 14 have built a language of admissible states and admissible continuations, and shown Rust's type system enforcing that language at compile time wherever it can. This chapter asks what should happen at the rare moments when that enforcement is not enough, when a program, despite the type system's best efforts, arrives at runtime at a state its own types claimed could never occur. The bridge and the pressure vessel suggest the answer: not a graceful, silent accommodation, but an immediate, visible stop, because everything computed afterward would rest on an assumption already known to be false.

15.2 General Principle: A Contradiction, Not an Ordinary Failure

Chapter 14 distinguished ordinary failure, the kind a `Result` represents, from absence, the kind an `Option` represents; both are anticipated outcomes, outcomes the function's own honest domain, in Chapter 12's sense, was designed to include. A panic is categorically different from either. It does not arise from an anticipated branch of a function's behavior; it arises when an assumption the program's types encoded as always true, an index within bounds, a division by a nonzero number, an `Option` already confirmed to be `Some`, turns out to be false at the exact moment the program relied on it.

This is why treating a panic as merely a more severe kind of `Result` would be a category error. A `Result`'s `Err` variant is a value the function's own logic anticipated and constructed deliberately; a panic is the discovery that the program's model of its own state, the very state space Chapters 6 and 7 showed being declared and enforced by the type system, has come apart from the actual state of the running program. Continuing execution past that point is continuing to compute using a model already known to be wrong, which is precisely the bridge-inspector's forbidden move: proceeding as though the falsified assumption had not been falsified.

15.3 Formal Definition: The Admissible Manifold and Departure From It

Borrowing the geometric language this chapter's title invokes, think of a program's full space of reachable states, across every type and every function, as an *admissible manifold*: the set of states the type system, together with every function's internal logic, jointly certifies as reachable through legitimate, type-checked execution. Every guarantee developed in Chapters 5 through 14, unique ownership, exhaustive matching, honestly widened codomains, is a constraint helping to define this manifold's boundary. A panic occurs at the exact instant execution would require leaving the manifold: performing an operation, indexing past a slice's length, unwrapping a confirmed-impossible `None`, dividing by a runtime-computed zero, for which no state on the manifold provides a defined continuation.

Formally, if $\mathcal{M} \subseteq \Sigma$ denotes the admissible manifold and $\sigma \in \mathcal{M}$ is the program's current state, a panic-triggering operation is one for which no $\sigma' \in \mathcal{M}$ exists satisfying the operation's own definition of a valid result. Rust's response, unwinding the stack or aborting the process rather than returning some arbitrarily chosen σ' outside $\mathcal{M}$, is a direct enforcement of Section 15.2's contradiction principle: rather than silently manufacture a state the type system never certified, the runtime halts, refusing to let execution continue on a manifold it can no longer vouch for.

15.4 Rust Mechanism: Panic as a Last, Loud Resort

```
fn get_config_value(values: &[i32], index: usize) -> i32 {
    values[index]
    // If index >= values.len(), this indexing operation has no
    // defined result within Σ_i32. There is no i32 the manifold
    // can certify as the answer, so the program panics rather
```

```
    // than returning an arbitrary, uncertified value.
}
```

Contrast this with the honest-domain vocabulary from Chapter 12: `get_config_value`'s claimed domain, any `usize`, is broader than its honest domain, indices less than `values.len()`, exactly the dishonest-signature pattern Chapter 12 warned against. A panic here is the runtime discovering, the hard way, a gap the type signature never admitted existed. The idiomatic remedy, as Chapters 13 and 14 already showed, is to close the gap in the types themselves, for instance by returning `Option<i32>` via `values.get(index)` instead of indexing directly, converting an unrepresented possibility into a represented one before it can ever reach the manifold's edge.

15.5 Worked Example: Choosing Between a Panic and a Represented Failure

```
fn get_config_value_total(values: &[i32], index: usize) -> Option<i32> {
    values.get(index).copied()
    // Claimed domain: any (values, index) pair.
    // Honest domain: the same, exactly, because the out-of-bounds
    // case is now represented as None rather than excluded from
    // the signature's truth. This function cannot panic.
}
```

This is precisely Chapter 13's strategy, applied retroactively to the panicking version from Section 15.4: the possibility that used to force a departure from the admissible manifold has been folded back into the manifold itself, as an ordinary, anticipated `None` rather than a contradiction. A function that never panics is not a function that got lucky; it is, in the vocabulary this chapter has built, a function whose entire claimed domain has been shown to lie within the manifold, with no operation inside it capable of demanding a state the manifold does not contain.

15.6 Interpretation Exercise: An Unwrap That Is Actually Justified

```
fn first_positive(values: &[i32]) -> i32 {
    let found = values.iter().find(|&&v| v > 0);
    if found.is_none() {
        panic!("caller violated precondition: no positive value present");
    }
    *found.unwrap()
}
```

This compiles and, given the surrounding `if`, the `.unwrap()` can never actually panic at the point it executes. The question worth asking is: *is this `.unwrap()` a departure from the admissible manifold, or is it something else?* It is not a contradiction in Section 15.2's sense, because the preceding `if` has already established, locally and verifiably by a human reader tracing the two branches, that `found` is `Some` by the time `.unwrap()` runs; the manifold was never actually at risk. What this example shows instead is that the type system cannot see this local

proof; `Option<i32>`'s type alone does not encode "already confirmed to be `Some`" the way `NonEmptyStr` from Chapter 12 encoded "already confirmed non-empty." The explicit `panic!` with a message describing a *precondition violation*, rather than a generic failure, is the author's way of documenting, for the next reader, that this particular unwrap is not a gap in the honest domain at all but a locally provable fact the type system simply lacks the vocabulary to certify on its own.

15.7 Connections: Closing Part IV

Part IV has developed a single, cumulative argument. Chapter 12 named the gap between a claimed domain and an honest one; Chapters 13 and 14 showed two ways of closing that gap by widening the codomain; this chapter has examined what happens at the exact boundary where the gap was never closed at all, the admissible manifold's edge, and why Rust's chosen response there, an immediate, visible halt, is the correct analogue of the bridge inspector's alarm rather than a defect to be tolerated. Part V turns to concurrency, where the admissible manifold this chapter has described is no longer maintained by a single thread of execution alone, and where the same question, what happens when an assumption the types encoded turns out to be false, reappears in a form where the falsification can originate from another thread entirely.

Invariant Established in This Chapter

A panic is the discovery, at runtime, that a state the type system certified as impossible has occurred; it is a contradiction, not an ordinary anticipated failure, and it should be treated as an immediate, visible departure from the program's admissible manifold rather than smoothed over. Wherever the gap it exposes can be closed in the types themselves, by narrowing an input or widening an output as Chapters 12 through 14 showed, it should be; where a panic remains after that effort, it should document the specific precondition whose violation it represents, since the type system's silence on that precondition is a limitation of its vocabulary, not evidence the precondition does not exist.

Part IV Summary

- Every function has an honest domain, narrower than its claimed domain, unless the gap is represented in the types (Chapter 12).
- Absence should be named as a distinct, compiler-checked point in a widened type, not patched informally (Chapter 13).
- Failure should be represented, with evidence, rather than hidden behind an undifferentiated signal (Chapter 14).
- A panic is the runtime discovery of a state the type system certified impossible, a contradiction rather than an anticipated outcome (Chapter 15).

Therefore: a sound program's types must account for every outcome a function can produce, including absence and failure; wherever they do not, the admissible manifold's edge is not eliminated, only deferred to a runtime that will eventually find it. Part V asks what becomes of these same guarantees once more than one path of execution is running at a time.

Part V

Concurrent Worlds

Chapter 16: Concurrency as Communication Geometry

From First Principles

Question: When two independent activities need to share something, is the hard problem what they share, or how it moves between them?

16.1 Phenomenon: Coordination Without a Single Timekeeper

Two kitchens in a large event catering operation, one at the venue and one at a central commissary, must coordinate a shared inventory of a scarce ingredient, without either kitchen able to see the other's actions as they happen. If both kitchens are simply told the current count and trusted to update it independently, a well-known failure emerges: both check the count, both see enough remaining, both commit to using it, and the combined total exceeds what was actually available, because each kitchen's belief about the shared quantity was formed at a moment that has since been invalidated by the other kitchen's own action. The problem was never that either kitchen made an error in isolation; it was that two independent activities updated a shared quantity without any mechanism ensuring their updates could not collide.

Air traffic control solves a structurally identical problem for aircraft approaching the same runway, and its solution is instructive: no aircraft acts on a private belief about the runway's availability. Every relevant action is mediated through a single controller who grants explicit, one-at-a-time clearance, and an aircraft without clearance simply does not have the authority to land, regardless of what it independently believes about the runway's state. The coordination problem was resolved not by giving every aircraft better information, but by restructuring who is allowed to act on that information, and when.

The same coordination problem, and the same range of solutions, recurs across many systems where several independent activities touch a shared resource. An airport's baggage handling system routes bags from dozens of simultaneously arriving flights onto a shared network of belts and chutes, and avoiding collisions between bags requires exactly the same discipline as avoiding collisions between database writes: something has to mediate access to each shared segment of belt. A factory assembly line coordinates dozens of workstations acting on the same product in sequence, each station's access to the product mediated by the line's own timing rather than left to chance. An orchestra conductor plays the same coordinating role air traffic control plays for aircraft, giving each section explicit, timed authority to enter rather than trusting every musician's independent sense of when to come in. Railway

signalling systems grant a single train exclusive authority over a block of track at a time, refusing a second train's entry until the first has cleared it, for exactly the reason two threads cannot both hold exclusive authority over the same value at once.

This chapter's title is not merely a figure of speech; it names a real design choice available to hardware as much as to software. Nearly all conventional digital circuits are synchronous, coordinated by a single global clock signal that tells every component when it is safe to assume every other component has finished its current step, a single timekeeper in the most literal sense. Karl Fant's work on Null Convention Logic develops an alternative: a clockless, delay-insensitive design discipline in which each computation signals its own completion locally, through the data itself, and the next stage proceeds only once it has actually received that signal, rather than once a fixed number of clock cycles has merely elapsed [3]. No global timekeeper coordinates the circuit at all; correctness follows entirely from each local handshake being honored, the hardware-level counterpart of this chapter's channels and mutexes, where a message's arrival or a lock's release, not a shared clock tick, is what licenses the next step to proceed.

Chapter 3 established that mutation invalidates the beliefs of any observer who reached a value beforehand, and that the cost of guaranteeing safety scales with how many paths can reach the value. Concurrency is the setting in which this cost becomes unavoidable rather than theoretical: two threads are, definitionally, two independent paths that can reach a shared value at genuinely the same moment, with no guarantee about which acts first. This chapter asks what changes, and what does not, when Chapter 3's single-threaded reasoning is stretched across multiple simultaneously executing paths.

What Goes Wrong Without This: Shared Mutable Queues

Before this chapter's communication geometries are introduced formally, it is worth seeing the failure they are designed to prevent: a queue shared directly between threads, with no discipline governing who may read or write it when.

```
// Hypothetical, not valid Rust:
// let queue = SharedQueue::new(); // one queue, no access discipline
// spawn_thread(|| { queue.push(job_a); });
// spawn_thread(|| { queue.push(job_b); });
// spawn_thread(|| { let job = queue.pop(); process(job); });
// -- if two pushes interleave at exactly the wrong moment inside the
// -- queue's own internal bookkeeping, the queue's length counter and
// -- its actual contents can disagree, corrupting the structure for
// -- every thread using it afterward
```

Each individual push or pop looks correct on its own; the danger is entirely in what happens when two of them execute at genuinely the same moment, updating the queue's internal state in an interleaving nobody anticipated. This is Chapter 3's mutation-invalidates-belief problem, arriving now in its concurrent form: two threads each believed they held enough authority to modify the queue, and nothing stopped both beliefs from being acted on simultaneously. The channels and mutexes this chapter develops exist specifically to make this interleaving impossible to accidentally write, rather than merely unlikely to trigger.

16.2 General Principle: Different Geometries, One Underlying Question

Threads sharing memory directly, channels passing messages between independent tasks, async futures suspending and resuming, and actors exchanging messages through mailboxes are usually taught as four separate topics, each with its own vocabulary and its own idioms. Underneath the differences in vocabulary, each is answering the same question Section 16.1's two kitchens and air traffic controller both faced: given that more than one independent activity might want to touch the same value, what geometry of communication ensures that no two activities can invalidate each other's beliefs about it at the same moment.

Call this a *communication geometry*: a specific answer to where authority over a shared value is allowed to be, how it can move between activities, and what an activity is permitted to assume about a value it does not currently hold authority over. Shared-memory threading is one geometry, where authority is negotiated through locks held for a bounded duration. Channels are a second geometry, where authority is transferred outright, ownership moving from sender to receiver the way Chapter 5 described ownership moving between single-threaded scopes. Async and actors are further variations on the same underlying question, examined in Chapter 18 and later sections. This chapter's task is to show that all of them are geometries for the same problem, not four unrelated mechanisms that happen to share the word "concurrency."

16.3 Formal Definition: Concurrent Reachability

Recall $R(v)$, the set of program paths capable of reaching a value v, from Chapter 5. In a single-threaded program, the paths in $R(v)$ execute in some definite, if not always obvious, sequential order, which is precisely what let Chapter 5's aliasing rules be checked once, at compile time, by reasoning about that order. In a concurrent program, two paths $p_1, p_2 \in R(v)$ may execute *interleaved*, with no total order between their individual operations on v guaranteed by the language alone. Call v *concurrently safe* if, for every possible interleaving of every $p_i \in R(v)$'s operations on v, Chapter 3's mutation-safety condition, that no path relies on a belief about v invalidated by another path's mutation, continues to hold.

This definition makes precise why concurrency is harder than Chapter 3's single-threaded case rather than merely a variation on it: single-threaded safety needs to be verified against one order of execution, while concurrent safety needs to be verified against every interleaving the scheduler might choose, a combinatorially larger space. A communication geometry, in Section 16.2's sense, is exactly a discipline that restricts this space of interleavings, or restricts what any single path is permitted to do within it, enough that safety can again be checked by local reasoning rather than by an infeasible case analysis over every possible schedule.

16.4 Rust Mechanism: The Type System Extends Across Threads

```
use std::sync::mpsc;
use std::thread;

fn produce_and_consume() {
```

```
    let (tx, rx) = mpsc::channel::<i32>();

    thread::spawn(move || {
        for i in 0..5 {
            tx.send(i).unwrap();
        }
        // tx is moved into this closure. The sending thread has
        // sole ownership of it, in exactly Chapter 5's sense,
        // for as long as this closure runs.
    });

    for received in rx {
        println!("got {received}");
    }
}
```

The `move` closure transfers ownership of `tx` into the spawned thread, using the same ownership mechanism Chapter 5 developed for single-threaded code, now crossing a thread boundary. Section 16.3's concurrent-safety question, whether any interleaving could produce a conflicting access to `tx`, does not need to be checked by reasoning about interleavings at all: since `tx` has exactly one owner after the move, and that owner is the spawned thread alone, no other path in the program retains `tx` $\in R(\cdot)$ to conflict with it. The channel converts a concurrency problem into an ownership-transfer problem, which Chapters 5 through 15 have already fully equipped the reader to reason about.

A channel of this kind is a familiar shape well outside software. A postal service transfers a letter's custody from sender to postal worker to recipient, one party fully responsible at a time, never two parties simultaneously entitled to open the same letter. A conveyor belt moves an item from one workstation's zone of responsibility to the next without either station needing to coordinate directly; the belt itself is the channel. Pneumatic tube systems, still used in some hospitals and older buildings, transfer a physical capsule's contents between locations with the same one-owner-at-a-time discipline, the capsule being unopenable, functionally, by more than one party during transit. A messenger carrying a sealed envelope between two offices is the same pattern again: the envelope's contents belong, at any given moment, to whichever party currently holds it, and the sealing itself enforces that no one along the route can silently inspect or alter what is inside before the intended recipient receives it.

16.5 Worked Example: A Geometry That Makes the Unsafe Case Impossible to Write

```
use std::sync::{Arc, Mutex};
use std::thread;

fn shared_counter() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
```

```
        handles.push(thread::spawn(move || {
            let mut guard = counter.lock().unwrap();
            *guard += 1;
            // Exclusive authority over the integer is granted only
            // for as long as `guard` is alive, exactly Chapter 5's
            // &mut authority, now checked at runtime by the mutex
            // rather than at compile time by the borrow checker.
        }));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

Five threads share access to the same integer through an `Arc<Mutex<i32>>`. `Arc` permits many owners simultaneously, in contrast to Chapter 5's single-owner rule, but only ever grants shared, read-only access to the `Mutex` it wraps; the `Mutex` itself is what reintroduces Chapter 5's exclusive-authority guarantee, now enforced at the moment `.lock()` is called rather than at compile time, because the compiler cannot know in advance which thread will acquire the lock first. This is Section 16.2's communication geometry made concrete: authority over the integer moves, one thread at a time, through the mutex, and Rust's type system refuses to compile any attempt to access the integer without going through it.

A mutex's everyday counterparts are all built around a single physical or procedural token that grants exclusive, temporary use. A single-occupancy bathroom's key, chained to a placard or held at a front desk, grants whoever holds it exclusive use of the room, with the next person waiting until the key is returned; two people cannot hold the same key at once, and the room's exclusivity follows directly from the key's. A one-lane bridge in a construction zone typically operates under exactly this discipline, with a flagger granting one direction of traffic exclusive use of the lane at a time and holding the other direction until it clears. A library's rare-book archive room, accessible to only one researcher at a time, protects the materials by making concurrent, uncoordinated handling structurally impossible rather than merely discouraged. A safety lock on industrial machinery, lockout-tagout in the vocabulary of workplace safety, grants whoever applied the lock exclusive authority to operate or service the machine, with the lock itself, not a sign or a verbal agreement, being what a second worker's attempt to start the machine must physically contend with.

Historical Note: The Therac-25

Between 1985 and 1987, a radiation therapy machine called the Therac-25 delivered massive radiation overdoses to at least six patients, several fatally, because of a race condition in its control software. An operator entering treatment parameters quickly, correcting an initial mistake within roughly eight seconds, could trigger a sequence in which the machine's high-power electron beam mode was already being configured by one part of the software while another part had not yet finished processing the operator's correction; the two processes shared mutable state describing the machine's configuration with no discipline governing which one's writes were authoritative at a given moment. The result, on the specific timing the race condition required, was the machine's most dangerous beam configuration firing without the

safeguards that configuration was supposed to require.

This is Section 16.1's shared-mutable-queue failure, with sensors and actuators standing in for the queue and a human life standing in for whatever the corrupted value goes on to affect. Two processes each believed they held enough authority to determine the machine's configuration, and nothing enforced Chapter 5's exclusive-authority rule between them; the interleaving that produced the fatal configuration was rare enough, roughly one specific eight-second timing window, that extensive testing before deployment never happened to trigger it. Ordinary safe Rust does not make race conditions of this shape merely less likely; Chapter 17 will show that they cannot be expressed at all without passing through `unsafe`, converting a timing-dependent tragedy into a compile-time refusal.

16.6 Interpretation Exercise: A Geometry That Looks Safe but Isn't

```
use std::sync::Mutex;

struct Account {
    balance: Mutex<i64>,
}

fn transfer(from: &Account, to: &Account, amount: i64) {
    let mut from_balance = from.balance.lock().unwrap();
    let mut to_balance = to.balance.lock().unwrap();
    *from_balance -= amount;
    *to_balance += amount;
}
```

Each individual field access here is protected by a mutex, and the code compiles. The question worth asking is: *does protecting each individual value make the interleaving space from Section 16.3 safe, or has the danger simply moved somewhere the type system cannot see?* If thread A calls `transfer(&alice, &bob, 10)` while thread B concurrently calls `transfer(&bob, &alice, 5)`, A may lock `alice`'s balance first while B locks `bob`'s balance first, and each thread then blocks waiting for the lock the other already holds: a deadlock. Both individual locks were acquired correctly, and Rust's type system, which only enforces exclusive authority over each protected value in isolation, has no vocabulary for the ordering relationship between two separate mutexes. This is the concurrency-era counterpart to Section 15.6's unwrap: a fact, here a required lock-acquisition order, that a human reasoner can establish but that lies outside what the type system was built to check on its own, and closing the gap requires an explicit discipline, such as always locking accounts in a fixed, globally agreed order, that the language does not enforce automatically.

Physiome in Practice

`physiome`, the author's whole-body physiology simulator introduced in Appendix A, gives this chapter's abstract communication geometries a concrete, documented setting to be checked against, rather than a merely plausible one. The simulator is organized around eleven organ subsystems, cardiovascular, renal, hepatic, immune, and others, each advancing on its own independent clock: cardiovascular ticks every simulated second, immune every five

minutes, renal every fifteen, reflecting genuine physiological timescales rather than a single global timestep forced on all of them. Cross-subsystem coupling is deliberately restricted to exactly one channel: a repair operator belonging to one subsystem may read fields from another subsystem's block of the shared state, and may write to any subsystem's block, but subsystem modules never call each other's functions directly. Two coupling instances are actually implemented this way: a hematologic coagulation operator reads the hepatic subsystem's `clotting_factors` field, so impaired liver function genuinely constrains how much the coagulation repair can improve; and the immune subsystem's fever response reads its own cytokine-level violation as the trigger but writes to the thermal subsystem's `core_temp` field as the effect.

This is Section 16.2's communication-geometry question, answered concretely rather than left hypothetical: cross-subsystem information is exchanged through the shared, immutable state itself, read and written at well-defined points, never through a direct call from one subsystem's code into another's, exactly the disciplined geometry Section 16.2 argued a shared resource needs and the shared-mutable-queue failure of Section 16.1 shows what happens without one.

16.7 Connections: Ownership as the Common Thread

Every communication geometry examined in this chapter, message-passing channels and lock-mediated shared memory alike, has turned out to be a variation on the same mechanism Chapters 5 through 15 already built: authority over a value is unique at any given moment, and every safe way of sharing that value concurrently is a disciplined way of moving that authority between paths, never duplicating it. Chapter 17 makes this connection explicit, examining why Rust's compile-time ownership guarantees, developed originally for single-threaded reasoning, turn out to be exactly the guarantees concurrent code needs, and why the language's ability to catch data races at compile time in ordinary safe code is a consequence of Part II's machinery rather than a separate feature layered on top of it.

Invariant Established in This Chapter

Authority may move but not duplicate. Every communication geometry examined in this chapter, channels, mutexes, and the variations Chapter 18 will add, is a disciplined mechanism for transferring or temporarily lending exclusive authority over a shared value between concurrently executing paths, never granting two paths conflicting authority over the same value at once. Concurrent safety is Chapter 3's single-threaded mutation-safety condition extended to hold across every possible interleaving, and a communication geometry is exactly what narrows that combinatorial space back down to something checkable, whether at compile time, as with ownership transfer, or at runtime, as with a mutex.

Chapter 17: Ownership and Fearless Concurrency

From First Principles

Question: If a guarantee was strong enough to rule out a bug on one thread, why would it need to be re-proven to rule out the same bug across many threads?

17.1 Phenomenon: A Guarantee That Was Already General

A building code requiring load-bearing walls to be built from materials rated for a certain weight was not written with any particular building in mind; it was written as a general constraint on what counts as a legitimate wall, applicable equally to a single-story cottage and to one floor of a forty-story tower. Nobody re-derives the load rating separately for each additional floor built on top of an already-compliant wall; the wall's rating was a property of the wall itself, verified once, and it continues to hold regardless of how much else gets built around it later.

A cryptographic signature scheme proven secure against forgery was not proven secure only for messages sent by a single person to a single recipient; the proof, if it is a real proof, holds for any number of participants sending any number of messages, because the proof never depended on how many parties were involved in the first place. Chapter 16 closed by observing that every communication geometry it examined turned out to be a disciplined way of moving or lending the same unique authority Chapters 5 through 15 had already fully specified. This chapter asks the question those observations were setting up: if ownership's guarantees were never actually stated in terms of a single thread to begin with, what work is left to do in order to make them hold across many threads at once?

17.2 General Principle: A Guarantee Proven Independent of Scheduling Is Proven for Every Schedule

Look back at how Chapter 5 actually stated its central guarantee: a value has exactly one owner at a time, and a mutable reference grants exclusive authority for as long as it is held, with no shared reference permitted to coexist. Nowhere in that statement, nor in the aliasing rule that Chapter 5's Section 5.5 built from it, does the argument depend on there being only one thread of execution. The guarantee is a claim about how many paths in $R(v)$ may hold authority over

v at a given moment, not a claim about how those paths happen to be scheduled, interleaved, or distributed across processor cores.

This is precisely the shape of the building code and the signature scheme from Section 17.1: a guarantee proven without reference to a particular number of participants, or a particular execution order, remains true under any number of participants or any execution order, because introducing more of either does not introduce any new premise the original proof failed to consider. Rust's aliasing rule was never a single-threaded guarantee that happened to also work for one thread; it was always a guarantee about aliasing, full stop, and single-threaded execution was simply the first, easiest-to-explain special case Chapter 5 used to introduce it.

17.3 Formal Definition: Interleaving-Independence

Recall Section 16.3's definition of concurrent safety: v is concurrently safe if, for every interleaving of every path in $R(v)$'s operations, Chapter 3's mutation-safety condition holds. Call a guarantee G about a value v *interleaving-independent* if G's proof references only the relationship between operations in $R(v)$, such as which operations require exclusive authority and which merely require shared observation, and never references a specific temporal order among them. Chapter 5's aliasing rule is interleaving-independent in exactly this sense: its proof establishes that no two conflicting authorities can coexist, a claim about the *structure* of $R(v)$ at any given moment, not about which moment a scheduler happens to choose.

This is what licenses the following, otherwise surprising, claim: an interleaving-independent guarantee proven for the single-threaded case is, without any additional argument, already a proof of Section 16.3's concurrent safety, because concurrent safety only demanded that the guarantee hold under every interleaving, and an interleaving-independent proof was never sensitive to interleaving in the first place. The work of Chapters 5 through 15 was not preparation for this chapter; it was, retroactively, already this chapter's proof, waiting for the concurrent setting to make its generality visible.

17.4 Rust Mechanism: `Send` and `Sync` as the Compiler's Own Audit

```
use std::rc::Rc;
use std::thread;

fn broken() {
    let shared = Rc::new(5);
    let shared2 = Rc::clone(&shared);
    thread::spawn(move || {
        println!("{shared2}");
    });
    println!("{shared}");
    // Does not compile: Rc<T> is not Send.
}
```

This fails to compile, and the reason is worth stating precisely rather than waved at. `Rc<T>` implements shared ownership using a reference count that is incremented and decremented without any synchronization, an implementation choice that is perfectly sound under Section

17.3's interleaving-independence only if every increment and decrement is guaranteed to happen on a single thread; move an `Rc` across a thread boundary and that guarantee is gone, and the count itself becomes exactly the kind of shared, unsynchronized mutable value Chapter 3 warned about. Rust encodes this fact directly in the type system: `Rc<T>` simply does not implement the `Send` marker trait, and the compiler refuses to move a non-`Send` value into a spawned thread's closure, catching the violation before the program ever runs rather than leaving it to be discovered as an intermittent, hard-to-reproduce count corruption.

17.5 Worked Example: The Same Data Structure, Corrected

```
use std::sync::Arc;
use std::thread;

fn fixed() {
    let shared = Arc::new(5);
    let shared2 = Arc::clone(&shared);
    thread::spawn(move || {
        println!("{shared2}");
    });
    println!("{shared}");
    // Compiles: Arc<T> uses atomic operations for its reference
    // count, restoring interleaving-independence, so it implements
    // Send wherever T itself does.
}
```

`Arc<T>` is, in every respect relevant to this chapter, the same shared-ownership idea as `Rc<T>`, with one change: its reference count is updated using atomic operations, instructions guaranteed by the hardware to behave correctly under any interleaving. That single change is enough to restore Section 17.3's interleaving-independence to the reference count itself, and the compiler reflects this precisely, by granting `Arc<T>` the `Send` implementation it withheld from `Rc<T>`. Nothing about ownership, ordinary Chapter 5 ownership, needed to change at all; only the one internal operation that was not yet interleaving-independent needed fixing.

17.6 Interpretation Exercise: A Type That Is Send but Still Needs a Lock

```
use std::sync::Arc;
use std::cell::Cell;
use std::thread;

fn still_broken() {
    let shared = Arc::new(Cell::new(0));
    let shared2 = Arc::clone(&shared);
    let handle = thread::spawn(move || {
        shared2.set(shared2.get() + 1);
    });
    shared.set(shared.get() + 1);
```

```
    handle.join().unwrap();
    // Does not compile: Cell<T> is not Sync, so Arc<Cell<i32>>
    // is not Send.
}
```

This fails to compile even though `Arc<T>` itself is `Send`. The question worth asking is: *what has this rejection actually detected, given that the previous section just established `Arc`'s safety?* `Cell<T>` permits interior mutation, changing a value through a shared reference, without any synchronization at all; it was designed for single-threaded use, where Chapter 5's aliasing rule is enforced entirely at compile time and no runtime coordination is needed. Wrapping it in `Arc` makes *sharing the wrapper* across threads possible, but does nothing to make *mutating the contents* through that shared wrapper safe, since `Cell`'s own mutation is exactly the unsynchronized, concurrently-unsafe operation Section 17.4's `Rc` example illustrated for reference counting specifically. Rust's `Sync` marker trait captures exactly this second, independent condition, safe to access from multiple threads through a shared reference, distinct from `Send`'s, safe to transfer ownership to another thread, and the compiler's refusal here is catching a genuinely different failure mode than Section 17.4's, not a redundant restatement of it.

Physiome in Practice

Chapter 16 described `physiome`'s organ subsystems as independently ticking, communicating through a disciplined geometry rather than shared mutable state. That design choice pays off directly in this chapter's terms. Every state transition in the simulator, whether a repair operator's correction or a subsystem's own advance, takes the current whole-organism state by immutable reference and returns a new state by value; the project's own documentation states plainly that no interior mutability exists anywhere in its eleven subsystem modules, and that this is checkable directly, by searching the source for `Cell`, `RefCell`, or `unsafe` and finding none, rather than merely asserted as an intention. An organ subsystem's update function is, in this chapter's vocabulary, interleaving-independent in the strongest possible sense: it depends only on the state it was explicitly given as an argument, never on some ambient value another subsystem might be concurrently modifying, because the language itself, not merely the project's discipline, makes uncontrolled mutation impossible to express.

This is what Section 17.3 calls a guarantee proven without reference to execution order: because each subsystem's correctness was established as a property of a pure function of its explicit inputs, that same proof, unmodified, already covers every interleaving a scheduler could choose between subsystems whose updates touch no shared mutable field, exactly Chapter 17's central claim that ownership's guarantees require no separate argument to extend to concurrent execution.

17.7 Connections: Closing the Argument Chapter 16 Opened

Chapter 16 showed several communication geometries, each a disciplined way of moving or lending authority across threads. This chapter has shown why those geometries were sufficient without any new theory of concurrency: Chapter 5's ownership and aliasing guarantees were interleaving-independent from the moment they were first stated, and the `Send` and `Sync` traits examined here are simply the compiler's mechanism for verifying, type by type, exactly where that independence does and does not yet hold, extending Chapter 9's

admissibility-contract machinery to a new pair of compiler-checked properties rather than introducing a separate concurrency-specific type system. Chapter 18 turns to async, where authority is not merely moved between threads but suspended and resumed across an interval of time in which the holding task may not be actively running at all, and asks what Chapter 5's continuation-validity guarantees, first developed for lifetimes, have to say about a continuation that can be paused.

Invariant Established in This Chapter

A guarantee proven without reference to execution order remains true under every execution order; Rust's ownership and aliasing rules, developed in Chapters 5 through 15 for reasoning that never actually depended on single-threaded scheduling, are already proofs of concurrent safety, not merely single-threaded safety that happens to also apply. `Send` and `Sync` are the compiler's mechanism for verifying, per type, exactly where this interleaving-independence holds and where an internal operation, such as an unsynchronized reference count or an unsynchronized interior mutation, has not yet been made safe to cross a thread boundary.

Chapter 18: Async as Deferred Continuation

From First Principles

Question: When a task is paused partway through, where does the part of the world it was still relying on go while it waits?

18.1 Phenomenon: Work Left Mid-Sentence

A surgeon interrupted mid-procedure by an emergency in another operating room does not simply walk away; the operating team stabilizes the patient's current state, notes exactly what has been done and what remains, and hands the room to a covering surgeon capable of resuming from that precise point rather than starting over. What makes the handoff possible is that the interruption was anticipated by the whole system: sterile fields, monitoring equipment, and documentation are all designed to preserve a mid-procedure state accurately enough that resumption is safe, not merely possible.

A bookmark in a book does something structurally similar for a much simpler case: it preserves exactly one fact, the page number, sufficient to resume reading later, while deliberately not preserving anything about the reader's mental state, attention, or the room they were sitting in. The bookmark's design reflects a judgment about what must be preserved for resumption to be meaningful, page position, and what can safely be discarded, everything else.

Suspending one activity to let another proceed, then resuming exactly where it left off, is a routine pattern well outside surgery. A diner who orders at a restaurant counter and sits down to wait, rather than standing at the register until the food is ready, has suspended the ordering interaction while the kitchen works, resuming it only once a number is called; nothing about the wait requires the diner to stand motionless in the meantime. Starting a load of laundry and doing homework while it runs suspends the laundry task at exactly the point where nothing further can happen until the machine finishes, without requiring the person to sit and watch the machine. Starting a large file download and continuing to edit a document does the same: the download proceeds independently, and the editing session is not blocked on it. A clinic that sends a patient for several independent tests, bloodwork, imaging, an EKG, and processes each as its own results arrive rather than waiting for all three to be drawn before starting any of them, is suspending and resuming several independent processes concurrently, exactly the shape async is built to express in software.

Chapters 5 through 15 examined a program's continuations, in Chapter 5's sense, as things that unfold within a single, uninterrupted stretch of execution: a borrow is valid for a lifetime, a lifetime is a span the compiler can check against the code around it. Async introduces a genuinely new complication: a continuation that can be paused, its progress preserved, and resumed later, possibly on a different thread, possibly after other code has run in between. This chapter asks what Chapter 5's continuation-validity machinery has to say once continuations are no longer required to run start to finish without interruption.

18.2 General Principle: A Continuation That Outlives Its Own Execution

Chapter 5 described a reference's lifetime as a proof that its validity span does not exceed its referent's actual span. That proof was implicitly built on an assumption every chapter since has quietly relied on: that the code holding the reference is actively executing, continuously, for the reference's entire validity span. An `async` function breaks this assumption directly. At an `.await` point, the function's execution is suspended, its local state, including any references it was holding, preserved somewhere, while the thread that was running it goes off to do other work entirely, possibly for an unbounded amount of time, before resuming.

This means an async function's continuation is not merely a proof about a span of code; it is a proof about a span of *time*, during which the function may not be actively running at all. The surgeon's handoff and the bookmark both worked because their designers decided, deliberately, what needed to be preserved across the interruption; an async runtime needs an equally deliberate answer, and Rust's answer, examined in the rest of this chapter, is to make the entire suspended state into an ordinary, ownable value rather than leaving it implicit in a call stack that has, in the interrupted sense, stopped existing.

18.3 Formal Definition: A Future as a Reified Continuation

An `async fn` in Rust does not execute its body when called; calling it produces a value implementing the `Future` trait, a state machine whose variants correspond to the function's possible suspension points, each variant carrying exactly the local state that must survive across that particular suspension. Where an ordinary function's continuation, in Chapter 5's sense, lived implicitly on the call stack for as long as the function ran, a `Future`'s continuation is reified: turned into an explicit value, of a concrete (if compiler-generated) type, that can itself be owned, moved, and stored, in exactly the sense Chapter 5 developed for any other value.

$$\texttt{Future} \approx \bigsqcup_i \Sigma_{\text{locals alive at suspension point } i}$$

a disjoint sum, in Chapter 7's sense, over the function's possible suspension points, each summand holding exactly the state that must be carried across that particular pause. Polling a future, the mechanism by which an async runtime resumes it, is a request to advance from whichever summand the future currently occupies to the next, exactly the same reconstruction-and-continuation pattern Chapter 8 described for an ordinary `match`, now spread out across time rather than compressed into a single synchronous call.

18.4 Rust Mechanism: Ownership as What Makes Reification Safe

```
async fn fetch_and_log(url: &str) -> String {
    let body = fetch(url).await; // suspension point: state includes `url`
    println!("fetched {} bytes", body.len());
    body
}
```

At the `.await` point, the compiler-generated future for `fetch_and_log` must carry `url` across the suspension, since it is still needed, at least implicitly through borrow-checking, for the function's remaining logic. This is exactly Chapter 5's lifetime machinery being asked a new question: not merely "does this reference remain valid for the textual span of code that uses it," but "does this reference remain valid for the possibly-much-longer span of wall-clock time during which the future might sit suspended before being polled again." The borrow checker answers this the same way it always has, by requiring that anything the future's state captures by reference outlive the future itself; this is why async functions holding references are so often awkward to write without the reference's lifetime being tied, explicitly, to something guaranteed to outlive the entire await.

18.5 Worked Example: Ownership Sidestepping the Lifetime Question Entirely

```
async fn fetch_and_log_owned(url: String) -> String {
    let body = fetch(&url).await;
    println!("fetched {} bytes from {url}", body.len());
    body
}
```

Taking `url` by value rather than by reference removes the lifetime question from Section 18.4 entirely: the future generated for this function owns its `String` outright, in Chapter 5's ordinary sense, and an owned value's validity is not a question of matching spans against some external referent at all, since there is no external referent to outlive. This mirrors a pattern Chapter 16 already introduced for threads: just as moving a value into a spawned thread's closure sidestepped the need to reason about cross-thread reference validity, moving a value into an async function's captured state sidesteps the cross-time reference validity question this chapter has been developing. Ownership, once again, converts a harder validity question into a simpler one by removing the external referent the harder question was about.

18.6 Interpretation Exercise: A Future That Captures More Than It Appears To

```
async fn process(data: &mut Vec<i32>) {
    let first = data[0];
    slow_network_call().await;
    data.push(first);
}
```

This compiles. The question worth asking is: *what does this future's reified state, in Section 18.3's sense, actually have to hold across the `.await`, and what does that requirement mean for anyone trying to use `data` concurrently while this future is suspended?* The future must carry the `&mut Vec<i32>` reference itself across the suspension point, since `data.push(first)` needs it after resuming; by Chapter 5's exclusive-authority rule, this means no other code anywhere in the program can access `data` at all, not merely while `process` is actively running, but for the future's entire lifetime, including however long it sits suspended waiting on `slow_network_call`. A caller who assumed the exclusive borrow was only held during `process`'s active computation, and expected the vector to be free for other use during the network wait, has misjudged exactly the time-extension Section 18.2 introduced: async does not shrink the span an exclusive borrow must be held for, it can dramatically lengthen it, since the borrow now spans however long the runtime takes to get around to polling the future again.

Physiome in Practice

`physiome` does not use `async fn` or `.await` anywhere; its scheduler is a discrete-event loop, not an async runtime. It is worth including here anyway, because its own architecture independently arrives at, and names, the exact idea this chapter has spent its length formalizing, built by hand rather than generated by the compiler. Every organ subsystem implements a `Continuation` trait exposing an `interval`, how much simulated time until it next wants to run, and an `advance` function computing its next state from the current one and an elapsed duration. A central scheduler tracks one next-fire time per subsystem and always advances whichever is soonest. This is Section 18.3's reification, performed explicitly rather than by a compiler transform: where an `async fn`'s suspended state is automatically packaged into a compiler-generated `Future`, a `physiome` subsystem's suspended state between one advance and the next is an ordinary, explicitly-held value the author's own code constructs and passes forward, the same underlying idea, a continuation whose state must be carried across an interval in which it is not actively running, arrived at independently and implemented by hand before ever reaching for the language feature this chapter is about.

The comparison is not merely decorative. The project's own account of a scheduling defect discovered during development, in which every subsystem's next-fire time was recomputed from the current global time on every iteration rather than tracked incrementally, meaning the fastest subsystem always appeared soonest and every slower one was starved indefinitely, is exactly Section 18.6's warning in a different guise: a continuation's state, here a next-fire time, must be the thing genuinely carried forward from one resumption to the next, not silently recomputed from an assumption that quietly stopped holding once more than one continuation was in play.

Connection to Category Theory

Readers with a category-theory background may notice that `async`/`.await`'s sequencing is monadic in the same sense as Chapter 14's `Result`: a `Future` supports composing suspended computations without unwrapping their suspension at every step, and `.await` plays a structurally similar role to `?`, propagating a computation's eventual value through a chain of dependent steps. This is offered as a side note for readers who already have the vocabulary [9]; nothing in this chapter's argument depends on it.

18.7 Connections: Closing Part V

Part V has traced a single argument to its conclusion. Chapter 16 showed concurrency's various geometries as disciplined ways of moving authority across simultaneously executing paths; Chapter 17 showed that Chapter 5's ownership guarantees required no new theory to cover this, because they were interleaving-independent from the start; this chapter has shown that async's central novelty, suspension across time rather than mere interleaving across threads, is handled by the same ownership machinery once a suspended computation's state is reified into an ordinary, ownable value whose lifetime obligations are exactly Chapter 5's, now measured against wall-clock suspension rather than textual scope alone. Part VI turns from the internal structure of a running program to its boundaries with the world outside any single function: the crate graph, the macro system, the unsafe escape hatch, and the foreign-function boundary, each examined as a different kind of edge at which this book's admissibility vocabulary must be either extended or deliberately, visibly suspended.

Invariant Established in This Chapter

A suspended computation's continuation must remain valid across the entire interval in which it is not actively running, not merely at the textual span its code occupies; Rust reifies this continuation as an ordinary, ownable `Future` value, and Chapter 5's ownership and lifetime rules apply to it unchanged, now measured against wall-clock suspension time rather than call-stack scope. An exclusive borrow captured across an `.await` is held for the future's entire suspended lifetime, not merely for its active computation, and this extension, not any new rule, is async's genuine novelty within the framework this book has built.

Part V Summary

- Every communication geometry, threads, channels, mutexes, is a disciplined way of moving or lending authority; authority may move but never duplicate (Chapter 16).
- A guarantee proven without reference to execution order remains true under every execution order, which is why ownership required no new theory to cover concurrency (Chapter 17).
- A suspended continuation must remain valid across its entire non-running interval, not merely its textual span; async is ownership's guarantees extended across time (Chapter 18).

Therefore: concurrency and asynchrony introduce no genuinely new theory of correctness; they are Part II's ownership guarantees, already general, extended first across simultaneity and then across suspended time. Part VI turns outward, to the boundaries a program crosses that lie partly or wholly outside this machinery's reach.

Part VI

Boundaries and Architecture

Chapter 19: Cargo as Ecosystem Architecture

From First Principles

Question: When you depend on someone else's code, what exactly are you promising your own users about it?

19.1 Phenomenon: Trust That Has to Travel Through Many Hands

A restaurant that sources its fish from a distributor is not merely buying fish; it is inheriting the distributor's own supply chain, including every judgment the distributor made about which boats to trust, how the catch was stored, and how long it sat before delivery. The restaurant's menu makes an implicit promise to diners, that the fish is fresh and safe, and that promise is only as good as the weakest link in a chain the restaurant did not fully see and cannot fully verify at the moment of serving. A single bad link anywhere upstream becomes, silently, the restaurant's own liability.

A construction contractor who subcontracts electrical work is in the same position: the general contractor's promise to the building's owner, that the building is safe and code-compliant, is only as strong as the subcontractor's own work, which the general contractor typically cannot re-verify in full without effectively redoing it. What makes this arrangement workable at all is not blind trust; it is a structure of licenses, inspections, and contracts that make each link's obligations explicit and checkable, so that trust can be extended without needing to be re-earned from scratch at every layer.

Chapter 9 described a trait as a contract that lets a caller depend on a promise without needing to know an implementation's internals. A dependency in a software ecosystem is the same idea at a much larger scale: pulling in a crate written by someone else is an act of extending trust across an organizational boundary, not merely a technical one, and the question this chapter asks is what structure, analogous to the licenses and inspections that make the restaurant's and the contractor's trust chains workable, makes that extension of trust something other than blind faith.

19.2 General Principle: A Dependency Graph Is a Trust Graph

Every crate a program depends on, directly or transitively, becomes part of that program's own admissible manifold in the sense Chapter 15 developed: code from a dependency exe-

cutes with the same authority as code written by the project's own authors, and a defect or a malicious change anywhere in the dependency graph is, functionally, a defect or a malicious change in the program itself. This is worth stating plainly because it is easy to treat a dependency as a black box whose internals simply do not matter, in the same spirit as Chapter 9's trait-bound callers; the difference is that a trait bound is checked by the compiler at every use, while a dependency's actual behavior is trusted largely on the strength of its version number and its reputation, not verified line by line before every build.

The practical consequence is that a package ecosystem's real architecture is not the crates themselves but the graph of trust relationships between them: which crates depend on which, how deep the transitive chain runs before reaching a crate no one has personally vetted, and what mechanism exists for a promise made by a crate at version 1.2 to still hold, or to be checked, at version 1.3. Cargo's tooling, examined in the rest of this chapter, is best understood as infrastructure for managing exactly this trust graph, not merely a build tool that happens to also fetch dependencies.

19.3 Semver as an Admissibility Contract Over Time

Chapter 9 formalized a trait as restricting a caller's admissible continuations to a fixed set of method signatures. Semantic versioning extends the same idea across time rather than across types: a crate's public API at version `1.x.y` is an implicit contract that any code written against `1.0.0` remains admissible against every later `1.x.y`, for any $x \geq 0$ and any y, so long as the major version does not change. A minor version bump promises new capability without breaking old promises; a patch bump promises no capability change at all, only internal repair; a major version bump is an explicit admission that the contract itself is being rewritten, and every downstream consumer's own admissibility with respect to the new version must be re-verified rather than assumed.

This is a genuinely different kind of guarantee from anything Chapters 1 through 18 examined, because it is not checked by the compiler in the way a trait bound or a borrow is. Semver is a promise the crate's author makes voluntarily, and Cargo's dependency resolution trusts that promise by default, resolving to the latest version satisfying a stated compatible range without re-verifying that the new version actually kept its word. The trust graph from Section 19.2 is, in large part, exactly this: a graph of unverified promises, each individually reasonable to trust, whose aggregate reliability the ecosystem depends on without any single participant able to fully audit it.

19.4 Cargo Mechanism: The Lockfile as a Pinned Trust State

```
[dependencies]
serde = "1.0"
tokio = { version = "1", features = ["full"] }
```

The version requirements in `Cargo.toml` express Section 19.3's semver contract directly: `"1.0"` means any `1.x.y` is acceptable, trusting every intermediate author's semver promise to have been honored. `Cargo.lock`, generated from this manifest, pins the exact resolved versions actually used in a given build, converting an open-ended trust relationship, "any

compatible version," into a closed, reproducible one, "exactly this version, verified to work together, until someone deliberately updates the lockfile." This is the same move Chapter 12 described for narrowing a claimed domain down to an honest one: the manifest's requirement is broader than what any single build actually needs, and the lockfile is what narrows it, deliberately, to a specific, checkable point.

19.5 Worked Example: A Transitive Dependency's Promise Silently Broken

Consider a project depending on crate `A` version `1.2`, which itself depends on crate `B` version `2.0`. If `B` releases `2.1` and `A`'s manifest accepts any `2.x`, Cargo may resolve to `B 2.1` on a fresh build without any change to the top-level project's own manifest at all. If `B 2.1` technically kept its public API compatible but subtly changed some behavior its semver promise did not cover, floating-point rounding, iteration order, timing, the top-level project inherits that change without a single line of its own code having been touched, and without `A`'s author having done anything wrong by the letter of Section 19.3's contract. This is the trust graph's structure made visible: the top-level project extended trust to `A`, `A` extended trust to `B`, and a change several links away can surface as a defect the top-level project's own authors have no direct way to anticipate.

19.6 Interpretation Exercise: A Lockfile That Looks Safe but Isn't

A team commits `Cargo.lock` to version control, believing this guarantees every developer and every deployment builds with identical dependencies. The question worth asking is: *what does the lockfile actually pin, and what does it leave unpinned?* `Cargo.lock` pins the resolved version of every crate in the dependency graph as of the last time it was regenerated, but it says nothing about the contents of the crate registry itself remaining unchanged; a yanked version, a registry outage, or, in principle, a compromised registry account able to re-publish under an already-used version number, all lie outside what the lockfile's pinning can protect against. The lockfile closes the gap Section 19.3 described between an open-ended semver range and a specific, reproducible build, but it does so by trusting that the specific version it names, once fetched, is the same bytes every time it is fetched again; that further trust is a separate claim, backed by registry infrastructure Cargo does not itself fully verify, and conflating "the lockfile is committed" with "the build is fully reproducible and tamper-proof" overstates what the mechanism actually guarantees.

19.7 Connections: Opening Part VI

Part VI turns from the internal structure of a single crate's code, which Chapters 1 through 18 examined in depth, to the boundaries at which a crate meets something outside its own admissibility guarantees entirely: other crates, in this chapter; the code-generation boundary crossed by macros, in Chapter 20; the compiler's own verification, deliberately suspended, in unsafe Rust, Chapter 21; and a language boundary crossed entirely, in FFI, Chapter 22. Each of these is a different kind of edge at which this book's vocabulary of admissibility, ownership,

and reachability either extends cleanly or must be explicitly, visibly re-established, and Cargo's dependency graph, examined here as a trust graph rather than a mere build mechanism, is the first and most pervasive of these boundaries, since nearly every nontrivial Rust program crosses it many times before a single line of its own logic runs.

Invariant Established in This Chapter

A dependency graph is a trust graph: every crate a program depends on, transitively, becomes part of that program's own admissible manifold, and semantic versioning is the voluntary, unverified-by-the-compiler contract that lets this trust be extended without re-auditing every dependency at every build. A lockfile narrows an open-ended semver range to a specific, reproducible resolution, closing the gap between claimed and honest compatibility for a given build, but it pins only the resolved version graph, not the deeper claim that a named version's contents remain identical wherever it is fetched from again.

Chapter 20: Macros as Controlled Code Generation

From First Principles

Question: If code can write code, who is responsible for verifying what gets written?

20.1 Phenomenon: Templates That Still Have to Be Checked

An architect's standard detail drawing for a stair railing is not itself a finished design; it is a template, parameterized by height and span, that a drafter instantiates for each specific staircase in a project. The template speeds up drawing enormously, since the underlying geometry does not need to be re-derived from scratch each time, but it does not remove the need for a structural review of each instantiated result: a railing height that violates code is still a violation, whether it was drawn freehand or generated by filling in a template's blanks. The template controls *how* the drawing is produced; it does not, by itself, guarantee the result is compliant.

A form letter generator used by a government agency, filling in a citizen's name, case number, and outcome into a fixed paragraph structure, speeds up correspondence dramatically, but a bug in the template, a field substituted in the wrong place, an off-by-one in which case details populate which blank, produces thousands of subtly wrong letters at the same rate it would have produced correct ones, precisely because the automation multiplies whatever the template actually says rather than what its author intended it to say.

Chapter 9 formalized a trait as a contract whose obligations are checked once, at the point an implementation is written, and relied upon thereafter without re-verification. A macro is a different kind of automation entirely: rather than checking a promise once, it generates new code, potentially different at every invocation, and that generated code must still satisfy every guarantee this book has developed, ownership, exhaustiveness, honest domains, exactly as if a human had typed it by hand. This chapter asks what it means for a mechanism that writes code to remain accountable to the same admissibility discipline the rest of the language enforces.

20.2 General Principle: Generation Time Is Not Verification Time

Chapter 12 distinguished a function's claimed domain from its honest domain. A macro introduces a third moment worth distinguishing alongside these two: the moment the macro

itself is *defined*, the moment it is *invoked* at a particular call site, and the moment the code it generates is actually *compiled*. A macro's author writes the template once, at definition time, with some intended pattern of correct usage in mind, in the same spirit as the architect's railing detail. But nothing about writing the template verifies anything; verification happens only later, when a specific invocation's generated code is type-checked, borrow-checked, and otherwise held to every standard this book has developed, exactly as if that code had been written directly by the macro's caller rather than produced mechanically.

This matters because it clarifies what a macro actually buys its user: not an exemption from any of the guarantees Chapters 1 through 19 established, but a mechanical shortcut for producing code that must still satisfy all of them. A macro that expands into code violating ownership, or into a non-exhaustive match, or into a dishonestly-typed function, does not get a pass because a macro produced it; the resulting compiler error simply appears at the expansion site, sometimes in a form harder to read than if the code had been written by hand, since the error now describes generated code the macro's caller never directly typed.

20.3 Formal Definition: A Macro as a Syntax-to-Syntax Function

A declarative macro, defined with `macro_rules!`, is a function not from values to values but from token trees to token trees: given input tokens matching one of its declared patterns, it produces a fixed template of output tokens, with the matched input substituted into designated positions. Call this $M : \text{Tokens} \to \text{Tokens}$. Crucially, M operates entirely prior to type-checking; it has no access to the types Chapters 6 through 11 spent so much effort declaring precisely, and no ability to consult Chapter 5's ownership rules while deciding what tokens to emit. M's only obligation, in the sense this book has used the word, is syntactic: that the tokens it emits, for every input the macro claims to accept, form syntax the compiler can subsequently attempt to admit through the ordinary battery of checks.

This is why a macro's own claimed domain, in Chapter 12's sense, the input patterns it declares itself willing to match, can be considerably broader than its honest domain, the inputs for which the generated code actually compiles: M can match tokens fine while still emitting code that later fails to type-check, fails to borrow-check, or fails exhaustiveness, and the failure surfaces not at M's own definition but at the specific call site whose particular substitution first exposed the gap.

20.4 Rust Mechanism: Hygiene as Reachability Preserved Across Expansion

```
macro_rules! swap_and_print {
    ($a:expr, $b:expr) => {
        {
            let temp = $a;
            $a = $b;
            $b = temp;
            println!("{} {}", $a, $b);
        }
    };
```

```
}
```

A naive, non-hygienic expansion of a macro like this would risk a variable named `temp` at the call site colliding with the macro's own internal `temp`, silently capturing or being captured by it, a violation of exactly the reachability discipline Chapter 5 spent an entire chapter establishing: the macro's internal binding and the caller's own bindings would become unintentionally aliased. Rust's macro hygiene mechanism prevents this by tracking, at the token level, which syntax originated inside the macro's own definition and which was supplied by the caller, ensuring identifiers from the two sources cannot accidentally collide even if they share a name. Hygiene is, in effect, Chapter 5's reachability guarantee extended to cover the macro expansion process itself: a caller's `temp` and the macro's own `temp` remain distinct paths in $R(\cdot)$, exactly as Chapter 5 would require of any other two same-named bindings in different scopes.

20.5 Worked Example: A Claimed Domain Broader Than the Generated Code Can Support

```
macro_rules! first_item {
    ($v:expr) => {
        $v[0]
    };
}

fn use_it(data: &[i32]) -> i32 {
    first_item!(data)
}
```

`first_item!` accepts any expression syntactically, its claimed domain in Chapter 12's sense is essentially unrestricted, but the code it generates, indexing at position zero, has exactly the honest-domain gap Chapter 12 and Chapter 15 both examined at length: called with an empty slice, the generated `data[0]` panics, leaving the admissible manifold in precisely the sense Chapter 15 formalized. The macro has not introduced a new kind of danger; it has simply mechanized the production of an old one, the same claimed-domain-exceeds-honest-domain gap that a hand-written indexing expression would have exhibited, now generated automatically at every call site that happens to invoke the macro.

20.6 Interpretation Exercise: A Macro That Hides Its Own Panic Behind Convenience

```
macro_rules! get_or_die {
    ($opt:expr, $msg:expr) => {
        match $opt {
            Some(v) => v,
            None => panic!($msg),
        }
    };
}
```

```
fn lookup(records: &[(String, i32)], key: &str) -> i32 {
    let found = records.iter().find(|(k, _)| k == key).map(|(_, v)| *v);
    get_or_die!(found, "key not found")
}
```

This compiles and, for a missing key, panics with the message `"key not found"`. The question worth asking is: *has this macro made the honest-domain gap from Section 20.5 disappear, or has it merely made it more convenient to invoke?* `get_or_die!` performs Chapter 8's exhaustive case analysis correctly, both `Some` and `None` are handled, so in the narrow sense of Chapter 8's exhaustiveness guarantee nothing has been skipped. But the `None` arm's handling is, structurally, identical to Chapter 15's admissible-manifold departure: `lookup`'s honest domain is still narrower than its claimed one, exactly as it would be if the `panic!` had been written out by hand at the call site, and the macro's convenience has, if anything, made this gap easier to introduce repeatedly across a codebase without each instance being individually scrutinized the way a hand-written panic might prompt a reviewer to ask "should this really panic here." A macro that makes an honest-domain gap more convenient to reproduce is not thereby making that gap safer; Chapter 12's remedies, Option or Result as the return type, remain exactly as applicable to macro-generated code as to hand-written code, and the macro's convenience is only a genuine improvement once it is applied to closing the gap rather than to producing more instances of it.

20.7 Connections: A Boundary Internal to the Language Itself

Where Chapter 19 examined a boundary between a program and code written by other people, this chapter has examined a boundary within a single author's own code: the line between the syntax a macro's definition commits to producing and the fully-checked code the compiler ultimately admits. Every guarantee this book has developed remains in force on the far side of that boundary; a macro changes when and how code is produced, never what that code is subsequently held to. Chapter 21 examines a genuinely different kind of boundary, one where a piece of code is deliberately exempted, in a narrow and explicitly marked way, from some of Chapter 5's own compile-time verification, and asks what discipline replaces the compiler's guarantee once it has been suspended on purpose.

Invariant Established in This Chapter

A macro is a function from syntax to syntax, not from values to values, and it earns no exemption from any guarantee this book has developed; the code it generates is checked exactly as if a human had written it by hand, at the specific call site whose substitution first exposed any gap. Hygiene extends Chapter 5's reachability discipline across the expansion boundary itself, keeping a macro's internal bindings and a caller's bindings from colliding even when they share a name; convenience of invocation is orthogonal to, and must never be mistaken for, closure of the honest-domain gaps Chapters 12 through 15 already showed how to close.

Chapter 21: Unsafe Rust as Explicit Boundary Crossing

From First Principles

Question: When a guarantee can no longer be checked automatically, does the guarantee stop existing, or does responsibility for it simply change hands?

21.1 Phenomenon: Zones Where the Usual Rules Are Suspended, Not Abolished

A hospital's sterile surgical field is governed by rules so strict that almost nothing is trusted without independent verification: instruments are counted before and after, packaging is checked for intact seals, anyone crossing the boundary scrubs in under observation. Deeper inside that field, during the operation itself, the surgeon works with a degree of direct, unmediated control that would be reckless anywhere else in the hospital, cutting tissue, moving organs aside, making judgment calls no checklist can fully anticipate. This is not a suspension of the hospital's commitment to patient safety; it is a deliberate, narrowly bounded region where safety is maintained by the surgeon's own trained judgment instead of by institutional checklists, precisely because the checklists cannot see finely enough into what is actually happening at that moment.

A test pilot flying an aircraft outside its certified envelope is in a structurally similar position: every ordinary safety margin built into commercial aviation, margins that exist precisely so pilots do not need to personally verify structural limits on each flight, has been deliberately set aside for this specific flight, replaced by the pilot's own expertise, instrumentation, and a support team monitoring in real time. The aircraft has not stopped needing to obey the laws of aerodynamics; it is simply that the certified envelope's built-in safety margin, which normally does that verification on the pilot's behalf, is not applicable here, and something else, more expensive and more failure-prone than a certificate, has to substitute for it.

The same shape of zone, ordinary automatic protections deliberately narrowed and replaced by certified, individual responsibility, governs a range of everyday high-risk work. Entering an active construction site requires a hard hat, a signed waiver, and often an escort, precisely because the site's hazards are real and the ordinary protections of a public sidewalk do not extend past its perimeter; entry is not an invitation to ignore the danger, it is a transfer of responsibility for managing it onto whoever enters. A laboratory handling hazardous

chemicals operates under similar terms: fume hoods, protective equipment, and documented handling procedures replace, inside the lab, the general environmental safety the building outside otherwise takes for granted. A high-voltage electrical cabinet is marked, locked, and entered only by someone qualified to work on live circuits, because the ordinary assumption that touching an electrical panel is safe simply does not hold past that door. Aircraft maintenance performed under a certified procedure grants the certified technician authority to bypass checks a passenger would never be allowed to bypass, precisely because the technician's certification is itself a documented claim that they, personally, can be trusted to maintain the safety the checklist would otherwise enforce. In every one of these cases, the barrier, the hard hat requirement, the fume hood, the locked cabinet, the certification, is not permission to disregard safety; it is a permit, granted specifically because the usual automatic protections cannot reach past it, and Rust's own `unsafe` keyword is best read the same way, as a permit rather than as permission to stop caring about correctness.

Chapters 5 through 20 have described a long chain of guarantees the Rust compiler checks automatically, ownership, exhaustiveness, honest domains, macro hygiene. The `unsafe` keyword marks a boundary structurally identical to the surgeon's sterile field or the test pilot's flight envelope: a narrowly scoped region where some of those automatic checks are deliberately suspended, not because the underlying guarantees stop mattering, but because the compiler's checking mechanism cannot verify them in this particular case, and something else must take over the verification instead.

21.2 General Principle: Suspension of Checking Is Not Suspension of Obligation

It is tempting to read `unsafe` as meaning "the rules do not apply here." Every example in Section 21.1 argues against that reading. What actually changes inside an `unsafe` block is narrower and more specific: the compiler stops automatically verifying a small, fixed set of additional operations, raw pointer dereferences, calls to other `unsafe` functions, access to mutable statics, implementing certain unsafe traits, that it would otherwise refuse to compile at all. Every other guarantee this book has developed, Chapter 5's aliasing rules for ordinary references, Chapter 8's exhaustiveness for ordinary matches, Chapter 12's honest-domain concerns for ordinary functions, continues to apply inside an `unsafe` block exactly as it did outside one; `unsafe` does not touch them.

What genuinely changes is who is responsible for the specific, narrow set of additional obligations the keyword unlocks. Outside `unsafe`, the compiler both states the obligation and verifies it automatically. Inside `unsafe`, for that narrow set of operations, the obligation still exists, a raw pointer dereference is still only sound if the pointer is valid and appropriately aligned, but verifying it becomes the programmer's job, discharged through reasoning the compiler cannot itself perform and simply has to trust was done correctly.

21.3 Formal Definition: A Locally Suspended Verification, Not a Locally Suspended Guarantee

Let $\mathcal{V}$ denote the full set of admissibility guarantees this book has developed: ownership uniqueness, aliasing exclusivity, exhaustiveness, and so on. Let $\mathcal{V}_{\text{auto}} \subseteq \mathcal{V}$ denote the subset the compiler verifies automatically in ordinary, safe code. `unsafe` does not shrink $\mathcal{V}$, the guarantees a sound program must actually uphold; it identifies a small set of additional operations O_{unsafe}, each with its own associated obligation, whose verification is removed from $\mathcal{V}_{\text{auto}}$ within the marked block and must instead be discharged by the programmer's own argument, an argument the compiler will neither construct nor check, but whose absence or falsity constitutes exactly the same kind of unsoundness Chapter 15 described for an ordinary panic, an operation performed outside its true admissible manifold, only now silent rather than caught.

This is why idiomatic `unsafe` code is typically documented with an explicit safety comment stating the argument the compiler cannot make on its own: the comment is not decoration, it is the discharge of an obligation that has not gone away, merely moved from the compiler's domain into the programmer's, and a future reader auditing the code needs that argument in order to re-verify, by hand, what the compiler is no longer verifying automatically.

21.4 Rust Mechanism: The Narrowest Possible Scope for the Suspension

```
fn split_first_unchecked(s: &[i32]) -> (&i32, &[i32]) {
    // SAFETY: caller of this private helper always passes a
    // non-empty slice, enforced by the two call sites in this
    // module, both of which check .is_empty() first.
    unsafe {
        let first = &*s.as_ptr();
        let rest = std::slice::from_raw_parts(s.as_ptr().add(1), s.len() -
            1);
        (first, rest)
    }
}
```

The `unsafe` block here is scoped as narrowly as the two raw-pointer operations that actually require it; everything else in the function, including the ordinary reference types in its signature, remains fully checked by the compiler exactly as Chapter 5 through Chapter 20 described. The safety comment states, in prose the compiler cannot verify, exactly the argument Section 21.3 described: which precondition, non-emptiness, is being relied upon, and why the author believes every call site actually satisfies it. This is the direct analogue of Section 21.1's surgeon documenting exactly what was done and why, so that anyone reviewing the case afterward, or resuming it, has the reasoning available rather than having to reconstruct it from scratch.

21.5 Worked Example: A Correctly Discharged Obligation

```
pub fn first_two(s: &[i32]) -> Option<(&i32, &i32)> {
    if s.len() < 2 {
        return None;
    }
    // SAFETY: s.len() >= 2 was just checked, so indices 0 and 1
    // are both in bounds, and get_unchecked's precondition holds.
    unsafe {
        Some((s.get_unchecked(0), s.get_unchecked(1)))
    }
}
```

This function is publicly callable with any slice at all; its claimed domain, in Chapter 12's sense, is genuinely honest, because the length check preceding the `unsafe` block establishes, and the safety comment states, exactly the precondition `get_unchecked` requires. The obligation Section 21.2 described, that the compiler no longer automatically verifies bounds for `get_unchecked`, has been discharged by an ordinary, fully-checked `if` statement immediately preceding it; the `unsafe` block's soundness rests on code the compiler *did* verify, which is the general pattern most correct `unsafe` code actually follows, a small unchecked operation resting on a larger scaffold of checked reasoning that the safety comment makes explicit.

Historical Note: Heartbleed

In 2014, a vulnerability in a widely deployed cryptographic library allowed an attacker to read up to sixty-four kilobytes of a server's memory per request, potentially including private keys, passwords, and other sensitive data, by sending a specially crafted heartbeat message. The heartbeat protocol let a client send a short payload along with a length field stating how long that payload was, and the server, replying, copied `length` bytes starting from the payload back to the client, trusting the attacker-supplied length rather than checking it against the payload actually received. A client could claim a length far larger than the payload it actually sent, and the server would copy whatever happened to sit in memory past the end of the real payload into its reply.

This is precisely Section 21.3's obligation, discharged nowhere. The operation, copying `length` bytes starting from a given pointer, has an unstated precondition, that `length` does not exceed what the caller actually provided, exactly the kind of precondition Rust's `unsafe` functions require a safety comment to state and the surrounding code to establish, as Section 21.5's `first_two` did with its preceding bounds check. Heartbleed's vulnerable code had no such check at all; the length field was used directly, with nothing resembling Section 21.4's narrowly-scoped, obligation-discharging `if` standing between the untrusted input and the memory copy. A Rust implementation reaching for `unsafe` to perform the same low-level copy would still need to write, and could not silently omit, exactly the length validation that was missing here; the language does not prevent an author from writing an equally careless safety comment, but it does insist that the comment, and the reasoning behind it, actually be written down at the point where the unchecked operation occurs, rather than assumed and never stated at all.

21.6 Interpretation Exercise: An Obligation That Looks Discharged but Moves

```
pub struct Cache {
    data: Vec<i32>,
}

impl Cache {
    pub fn get(&self, index: usize) -> &i32 {
        // SAFETY: index is always valid because callers are
        // expected to check bounds before calling this method.
        unsafe { self.data.get_unchecked(index) }
    }
}
```

This compiles, and the safety comment reads, superficially, like the discharge pattern Section 21.5 demonstrated. The question worth asking is: *has this obligation actually been discharged, or has it merely been described and then handed to someone who was never told about it?* Unlike Section 21.5's `first_two`, where the bounds check happens inside the same function as the unsafe operation, `Cache::get` is a `pub` method whose precondition, that `index` is in bounds, is enforced nowhere the compiler can see and stated nowhere the method's own signature communicates; `fn get(&self, index: usize) -> &i32` looks, from the outside, exactly like a function whose honest domain matches its claimed one, the same dishonest-signature pattern Chapter 12 first identified, now compounded by the fact that violating it does not merely panic, as Chapter 15's ordinary out-of-bounds indexing would, but produces genuinely undefined behavior, since `get_unchecked` performs no runtime check at all. The safety comment's claim, that "callers are expected to check bounds," is not a discharge of the obligation Section 21.3 described; it is a relocation of the obligation to every current and future caller of a public method, none of whom can see this comment from the call site, and Chapter 20's warning about macros generating honest-domain gaps applies here in an even sharper form: an `unsafe` operation's precondition, left undischarged and merely asserted in a comment on the wrong side of a public boundary, is a soundness bug waiting on the next caller who does not happen to read the implementation.

21.7 Connections: The Sharpest Boundary Yet

Chapters 19 and 20 examined boundaries where this book's guarantees continued to apply automatically, across a dependency graph and across macro expansion respectively, verified by the compiler on the far side exactly as on the near side. This chapter has examined the one boundary within safe-looking code where that automatic verification is deliberately, locally suspended for a narrow, fixed set of operations, with the obligation to uphold Chapter 5's aliasing rules, Chapter 12's honest domains, and every other guarantee this book has built shifted onto a human argument the compiler will not check. Chapter 22 extends this same discipline across a boundary the compiler cannot see into at all, the foreign function interface, where an entire other language's code, with no relationship to Rust's type system whatsoever, must be reasoned about using exactly the tools this chapter has introduced.

Invariant Established in This Chapter

Unsafe Rust is explicit boundary crossing, not blanket exemption: it suspends automatic compiler verification for a narrow, fixed set of operations, and every other guarantee this book has developed remains in force, checked exactly as before. The obligations attached to those narrow operations do not disappear when the compiler stops checking them; they move to the programmer, and are only genuinely discharged when the reasoning behind them is both actually sound and actually visible to whoever the obligation has moved to, not merely stated in a comment on the wrong side of whatever boundary hides it from the caller who most needs to see it.

Chapter 22: FFI as Interface Topology

From First Principles

Question: When two systems that speak entirely different languages must exchange something, what has to be true on both sides for the exchange to mean the same thing to each?

22.1 Phenomenon: Translation That Has No Referee

Two countries whose currencies are not directly convertible on any exchange must route a transaction through an intermediary currency, and the meaning of "the same amount of value" has to be reconstructed at each conversion, since neither country's banking system can see or verify the other's internal accounting directly. If either side's conversion rate is even slightly wrong, or if the intermediary currency's own value shifts between the two conversions, both parties can walk away from the transaction believing, correctly according to their own records, that fair value was exchanged, while the total amount of value has silently grown or shrunk in the crossing.

A power grid interconnection between two national systems running at slightly different frequencies requires a synchronizing device at the junction, one whose entire job is to reconcile two signals that are individually well-behaved on their own sides but meaningless to connect directly without an intermediary that understands both. Neither national grid's own internal safety mechanisms extend across the interconnection automatically; the synchronizing device is where an entirely new set of guarantees has to be established, from scratch, specific to the junction itself.

Chapter 21 examined a boundary where Rust's own compiler simply stopped checking a narrow set of operations, with every other guarantee remaining automatically enforced on both sides. A foreign function interface is a boundary of the second kind: the code on the other side is not Rust at all, has no relationship to Chapter 5's ownership rules, Chapter 8's exhaustiveness, or any other guarantee this book has built, and nothing about Rust's type system extends across the boundary automatically, the way neither national grid's safety mechanisms extend across the interconnection without the synchronizer's deliberate work.

22.2 General Principle: A Boundary Where Every Guarantee Must Be Rebuilt Locally

Chapter 19 examined a boundary, between crates, where the guarantees this book has developed continue to hold automatically on both sides, verified by the same compiler using the same rules. Chapter 21 examined a boundary, into `unsafe` code, where a narrow set of operations loses automatic verification but every other guarantee remains in force. An FFI boundary is qualitatively different from both: the code on the far side was not compiled by rustc, does not share Rust's notion of ownership, may have entirely different assumptions about null pointers, thread safety, or memory layout, and the compiler has no visibility into it whatsoever. Every guarantee this book has developed, if it is to hold at all on the far side of an `extern` boundary, must be re-established by an explicit, human argument at the exact point of crossing, in precisely the spirit of Chapter 21's safety comments, now applied to an entire foreign system rather than a single Rust operation.

This is not a minor extension of Chapter 21's discipline; it is Chapter 21's discipline pushed to its limit, because the party on the other side of an FFI boundary is not merely un-verified Rust code, it is a system that may not even share Rust's assumptions about what a valid value looks like. A Rust `bool` is required to be exactly `0` or `1` at the bit level; nothing on the C side of a boundary is obligated to respect that requirement, and a foreign function that hands back an arbitrary byte where Rust expects a `bool` has violated an invariant Chapter 6 would have called part of the type's declared shape, invisibly, from outside the system that declared it.

22.3 Formal Definition: The Interface Topology of a Boundary Crossing

Call the *interface topology* of an FFI boundary the complete set of assumptions that must hold on both sides for a value crossing it to retain its meaning: matching memory layout, matching calling convention, matching assumptions about ownership and who is responsible for eventual deallocation, matching assumptions about which values are even valid instances of the type being exchanged. Where Chapter 6 showed a struct's own fields fully determining its state space Σ_{struct} within Rust, an FFI boundary asks a harder question: does the foreign side's representation of "the same type" actually correspond, bit for bit and assumption for assumption, to Σ_{struct}, or does it merely resemble it closely enough to compile without complaint.

Rust's `#[repr(C)]` attribute is a direct response to this question: an ordinary Rust struct's memory layout is deliberately left unspecified by the language, free to be reordered or padded however the compiler finds most efficient, precisely because nothing outside the Rust compiler is meant to depend on it. `#[repr(C)]` is a promise, checked at compile time on the Rust side only, that the struct's layout will match the C language's own standardized layout rules, restoring exactly the shared assumption Section 22.2 identified as necessary for the crossing to be meaningful. Nothing about `#[repr(C)]` can verify that the foreign code was actually compiled against a matching declaration; that half of the interface topology remains, irreducibly, a human obligation.

22.4 Rust Mechanism: Declaring the Foreign Side's Shape by Hand

```
#[repr(C)]
pub struct Point {
    x: f64,
    y: f64,
}

extern "C" {
    fn distance(a: *const Point, b: *const Point) -> f64;
}

pub fn safe_distance(a: &Point, b: &Point) -> f64 {
    // SAFETY: Point is repr(C) and matches the layout the C
    // library's `distance` function expects; both pointers are
    // derived from valid Rust references and remain valid for
    // the duration of this call.
    unsafe { distance(a as *const Point, b as *const Point) }
}
```

The `extern "C"` block is Rust's mechanism for declaring, by hand, exactly the shape Section 22.3 described a foreign function as having; nothing about this declaration is checked against the actual C library's source, since the compiler has no access to it. If the real `distance` function's C declaration disagreed with this one, in argument types, in calling convention, in anything, the mismatch would produce undefined behavior at the call site with no compiler diagnostic at all, exactly the silent-obligation-relocation Chapter 21 warned about, now with the party the obligation has moved to being an entire external toolchain rather than a future Rust caller.

22.5 Worked Example: Ownership Reconstructed Across the Boundary

```
extern "C" {
    fn allocate_buffer(size: usize) -> *mut u8;
    fn free_buffer(ptr: *mut u8);
}

pub struct ForeignBuffer {
    ptr: *mut u8,
    len: usize,
}

impl ForeignBuffer {
    pub fn new(size: usize) -> Self {
        // SAFETY: allocate_buffer either returns a valid pointer
        // to `size` bytes or null; the null case is not yet
        // handled here and is examined in Section 22.6.
        let ptr = unsafe { allocate_buffer(size) };
        ForeignBuffer { ptr, len: size }
```

```
    }
}

impl Drop for ForeignBuffer {
    fn drop(&mut self) {
        // SAFETY: self.ptr was obtained from allocate_buffer and
        // has not been freed elsewhere, since ForeignBuffer's own
        // API provides no way to extract or duplicate the pointer.
        unsafe { free_buffer(self.ptr) };
    }
}
```

`ForeignBuffer` reconstructs Chapter 5's ownership guarantee across the FFI boundary entirely by hand: the foreign allocator has no notion of Rust ownership at all, it simply hands back a raw pointer, and `ForeignBuffer`'s own `Drop` implementation is what re-establishes, on the Rust side, the single-owner discipline Chapter 5 originally got for free from the borrow checker. This is Section 22.2's general principle made concrete: nothing about Chapter 5's guarantees crossed the boundary automatically, they were rebuilt, deliberately, using exactly the tools, a wrapper type and a `Drop` implementation, that Rust provides for reconstructing ownership discipline over a resource the compiler cannot see into.

22.6 Interpretation Exercise: A Reconstruction With a Silent Gap

The safety comment in `ForeignBuffer::new` names, explicitly, a case it does not yet handle: `allocate_buffer` returning a null pointer, presumably to signal an allocation failure. The question worth asking is: *what is the actual honest domain, in Chapter 12's sense, of the `ForeignBuffer` that results from this constructor when allocation fails?* If `allocate_buffer` returns null, `ForeignBuffer`'s `ptr` field becomes null while its type, `*mut u8`, gives no compile-time indication that this is possible; every later use of `self.ptr`, including the `free_buffer` call in `Drop`, silently inherits an unstated precondition, that the pointer is non-null, exactly the same dishonest-signature pattern Chapter 12 identified and Chapter 21 sharpened into undefined-behavior territory, now sitting at the exact junction Section 22.2 described as having no automatic verification on either side. The honest fix is the one this book has used throughout: `ForeignBuffer::new` should return `Option<Self>` or `Result<Self, AllocError>`, folding the null case into the type the way Chapter 13 folded absence into `Option<T>`, so that the FFI boundary's own failure mode, entirely foreign to Rust's type system on its own, is translated into an ordinary, honestly-typed Rust value the moment it crosses.

22.7 Connections: Closing the Boundaries of Part VI

Chapters 19 through 22 have examined a progression of boundaries, ordered by how much of this book's automatic verification survives the crossing: a dependency graph, where verification continues automatically but trust is extended on the strength of an unverified promise; a macro expansion, where verification continues automatically but the code being verified was generated rather than written; an `unsafe` block, where verification is suspended for a narrow, explicitly marked set of operations; and now FFI, where verification does not extend at all, and every guarantee this book has built, ownership, honest domains, even the basic validity of a

type's own bit pattern, must be reconstructed by hand at the exact point of crossing. Chapter 23 turns from correctness boundaries to a different kind of boundary this progression has been quietly assuming throughout, the boundary between a program's logical structure and its physical execution cost, asking what it means for fast code and correct code to so often turn out to be the same code.

Invariant Established in This Chapter

An FFI boundary carries none of this book's guarantees across automatically; the foreign side shares no relationship with Rust's type system, ownership discipline, or validity assumptions, and every one of these must be explicitly, manually reconstructed at the exact point of crossing, using the same tools, wrapper types, `Drop` implementations, and honestly-typed translations of foreign failure modes, that this book has developed throughout. The interface topology of a boundary crossing is only as sound as the weakest unstated assumption on either side, and unlike every boundary examined in Chapters 19 through 21, no compiler on either side is in a position to catch a mismatch automatically.

Chapter 23: Performance as Preservation of Locality

From First Principles

Question: When correct code and fast code turn out to be the same code, is that a coincidence, or a symptom of something they were both measuring all along?

23.1 Phenomenon: Two Complaints With One Root Cause

A warehouse manager who complains that pickers spend too much time walking, and a separate auditor who complains that inventory counts are frequently wrong, might appear to be raising two unrelated issues, one about efficiency, one about correctness. In many poorly laid out warehouses, both complaints trace back to the same root cause: items that are logically related, frequently picked together, or easily confused with one another are stored far apart or awkwardly interleaved. The walking time and the counting errors are two different symptoms of a single underlying failure, a layout that does not respect which items actually belong near each other, either physically or conceptually.

A poorly organized filing system produces the same double symptom in an office: retrieving a document takes longer than it should, and documents are frequently misfiled or lost, because the organizational scheme that would have made retrieval fast, grouping related documents together, is the same scheme that would have made misfiling obvious and rare. A filing clerk who only optimized for retrieval speed, without addressing the underlying organization, would find accuracy improving as a side effect, not because accuracy was separately pursued, but because both problems were symptoms of the same disorganization.

Chapter 4 named locality as a general principle: correct reasoning about a component should require only a bounded context. This chapter asks what happens to that same principle when the question shifts from "can a human reason about this code correctly" to "does this code run fast," and the central claim is that the warehouse manager's and the auditor's complaints are, in Rust, unusually often the same complaint, because both correctness and performance in this language trace back to the same underlying resource: how tightly a piece of code's actual behavior is bound to a small, predictable neighborhood of memory and control flow.

23.2 General Principle: Locality Violations Cost Twice

Chapter 4 measured locality in terms of a function's reasoning footprint, $N(f, v)$, the context a human reader needs to verify correctness. Modern hardware measures something structurally similar, though for a different reason: a processor's cache hierarchy is fast for data recently accessed or physically nearby in memory, and slow for data that must be fetched from far away, whether "far away" means a different region of RAM or, worse, a value behind a pointer requiring a fresh memory access to chase. A program whose data access pattern is unpredictable or scattered pays a real, measurable cost in cache misses, exactly mirroring the unbounded reasoning cost Chapter 4 described for a human reader forced to track state scattered across an entire program.

This is not a metaphorical similarity; it is frequently the same underlying cause wearing two different faces. A data structure built from many small, independently heap-allocated nodes linked by pointers, a naive linked list, say, or a tree of individually boxed nodes, forces both a human reader and the processor's cache to chase references outward from a bounded neighborhood into unpredictable, scattered locations: the human loses the bounded reasoning footprint Chapter 4 championed, and the processor loses the spatial locality its cache was designed to exploit. Fixing the layout to keep related data contiguous, a flat array instead of a scattered linked structure, often improves both properties simultaneously, not because the fix was aimed at both, but because both symptoms shared the root cause Section 23.1's examples illustrated.

23.3 Formal Definition: Access Locality as a Cousin of Reasoning Locality

Recall Chapter 4's formal locality condition: a property P of a component c is locally verifiable if some bounded neighborhood $N(c)$, independent of the surrounding system's size, suffices to verify P. Define, in parallel, the *access locality* of a computation as the degree to which the memory addresses it touches over some bounded time window are themselves clustered into a small, predictable neighborhood, rather than scattered unpredictably across the address space. Both definitions share the same shape: a claim that a bounded, nearby context suffices, for verification in Chapter 4's case, for fast retrieval in this chapter's case, and both degrade in the same direction when a design scatters what should have stayed together.

This parallel explains, without needing any additional principle, why so much of idiomatic, performant Rust looks identical to idiomatic, easy-to-verify Rust: preferring an owned, contiguous `Vec<T>` over a boxed, pointer-linked structure serves Chapter 4's reasoning-footprint goal, since a reader can trace the whole structure's lifetime through one owner, and serves access locality simultaneously, since the elements sit contiguously in memory. The two goals are not in tension needing to be balanced against each other; in a large fraction of ordinary code, they are the same goal, viewed from two different disciplines.

23.4 Rust Mechanism: Ownership as Compiler-Enforced Contiguity

```
struct LinkedNode {
    value: i32,
```

```
    next: Option<Box<LinkedNode>>,
}

struct FlatList {
    values: Vec<i32>,
}
```

`LinkedNode` scatters its elements across however many separate heap allocations `Box` performs, one per node, each potentially anywhere in the address space the allocator happened to place it; traversing the list means chasing a fresh, unpredictable pointer at every step, exactly the access-locality failure Section 23.3 described. `FlatList` stores every element contiguously within a single `Vec`'s backing allocation. Nothing about Rust's ownership rules forces this choice, both compile and both are memory-safe, but Chapter 5's ownership discipline is what makes `FlatList`'s contiguous layout easy and safe to express directly: a single `Vec` owns all of its elements outright, with no need for the shared, individually-boxed ownership a naive linked structure requires once more than one part of a program might want to reach into the middle of it.

23.5 Worked Example: A Change That Improves Both Symptoms at Once

```
struct Employee {
    name: String,
    department: String,
    salary: f64,
}

fn total_payroll_scattered(employees: &[Box<Employee>]) -> f64 {
    employees.iter().map(|e| e.salary).sum()
}

fn total_payroll_flat(employees: &[Employee]) -> f64 {
    employees.iter().map(|e| e.salary).sum()
}
```

`total_payroll_scattered` iterates over a slice of individually heap-allocated `Employee` records, each access to `e.salary` requiring a pointer chase to wherever that particular `Box` happened to be allocated. `total_payroll_flat` iterates over records stored contiguously. Both functions are equally easy to read and equally correct in Chapter 4's sense of local reasoning, since the difference here is purely representational, but only the second actually exploits Section 23.3's access locality; a profiler measuring the two against a large employee list would typically find the flat version substantially faster, for a reason that traces back not to any cleverness in the summing logic, identical in both versions, but entirely to how the underlying data was laid out before the loop ever started.

23.6 Interpretation Exercise: A Local-Looking Optimization That Breaks the Wrong Kind of Locality

```
struct Cache {
    data: std::collections::HashMap<u32, f64>,
}

impl Cache {
    fn get_or_compute(&mut self, key: u32) -> f64 {
        if let Some(&v) = self.data.get(&key) {
            return v;
        }
        let computed = expensive_computation(key);
        self.data.insert(key, computed);
        computed
    }
}
```

This memoization pattern is a standard, idiomatic performance optimization, and it compiles cleanly under Chapter 5's borrow rules. The question worth asking is: *has this optimization improved access locality in Section 23.3's sense, or has it improved a different kind of locality, temporal rather than spatial, while potentially worsening the spatial kind?* A hash map's entries are, by design, scattered across its backing storage according to each key's hash, deliberately *not* contiguous, since contiguous storage keyed by an arbitrary hash would defeat the map's own lookup strategy. `get_or_compute` trades one locality property, avoiding redundant computation by exploiting temporal locality, the same key is often requested again soon, for a data layout that is, in Section 23.3's spatial sense, considerably less local than a flat array would have been. This is not a mistake; redundant computation avoided is very often worth more than cache-friendliness lost, but it is worth naming precisely, because Chapters 4 and this chapter have been building toward a claim, that correctness-locality and performance-locality usually align, and a caching layer is exactly the case where two different kinds of locality, temporal and spatial, can be legitimately traded against each other rather than both improving together, and a reader who has internalized only the earlier chapters' alignment might mistakenly expect this trade to be free.

23.7 Connections: The Alignment Was Never a Coincidence

This chapter has argued that Chapter 4's reasoning-locality principle and a processor's own cache-locality preference are, in a large fraction of idiomatic Rust code, the same underlying property viewed through two different lenses, which is why writing code that a human can verify with a small, bounded context so often turns out to also be the code a processor can execute efficiently. Section 23.6's exercise showed the limit of this alignment: temporal and spatial locality can diverge, and knowing which kind of locality a given optimization actually targets, rather than assuming all forms of locality move together, is itself part of the discipline this book has been developing since Chapter 4.

A structurally similar alignment appears in signal and image processing, in the theory of sparse and redundant representations. Michael Elad's treatment of the field shows that a sig-

nal admitting a sparse representation, one expressible as a combination of very few elements from a suitably chosen basis or dictionary, is simultaneously easier to store efficiently, easier to reconstruct accurately from partial or corrupted data, and easier to reason about analytically, with all three properties following from the same underlying sparsity rather than being independently engineered [11]. The parallel to this chapter's argument is close enough to be worth naming directly: a data layout that is easy for a human to verify locally and a data layout that is efficient for a cache to access are, like a sparse representation's storage efficiency and its reconstructibility, two consequences of one and the same underlying structural property, not two separate goals that happen to have been jointly satisfied by luck.

The two chapters that close this book turn outward: Chapter 24 examines networking as a composition of the boundary crossings Part VI has already treated individually, and Chapter 25 turns to testing, asking how the admissibility guarantees this entire book has developed, many of them checked once at compile time, are verified for the remaining cases, the ones that depend on runtime behavior no static check can fully anticipate.

Invariant Established in This Chapter

Performance and correctness in Rust frequently trace back to the same underlying property, locality, viewed through two different lenses: a bounded reasoning footprint for a human reader, and a bounded, contiguous access pattern for a processor's cache. Rust's ownership discipline makes contiguous, single-owner data layouts easy and safe to express directly, which is why idiomatic-correct code and idiomatic-fast code so often coincide; where a design trades spatial locality for temporal locality, as a cache does, the trade should be made deliberately and by name, not assumed to be free on the strength of this chapter's general alignment.

Chapter 24: Networking as Composed Boundaries

From First Principles

Question: A networked service touches a socket, a wire format, and a database in a single request. Is that three separate correctness problems, or one problem crossed three times?

24.1 Phenomenon: A Single Trust Failure, Multiplied by Every Layer It Touches

A shipping company's parcel passes through a loading dock, a customs inspection, and a delivery van before it reaches its recipient, and a failure at any single stage, a mislabeled dock, a rejected customs form, a van that never arrives, is sufficient to fail the whole delivery, regardless of how well the other two stages performed. The company does not treat these as three independent reliability problems to be separately optimized; it treats the parcel's journey as a single chain, and the chain's reliability is bounded by its weakest link, not averaged across all three.

A hospital's medication order passes from a physician's prescription, through a pharmacy's verification, to a nurse's administration at the bedside, and a single transcription error at any of the three handoffs can produce a patient harm regardless of how carefully the other two stages were executed. Hospitals accordingly design cross-checks at each handoff specifically, not because any one stage is unusually failure-prone, but because a multi-stage process's overall trustworthiness is a property of its weakest crossing, not its average one.

A networked Rust service handling a single incoming request typically crosses at least three boundaries of exactly this kind before it can respond: the network socket, where bytes arrive with no guarantee they represent anything meaningful at all; the wire format, where those bytes are parsed into a structured value that may or may not conform to what the service expects; and a data store, where a query may fail, return nothing, or return something the service's types did not anticipate. Chapters 19 and 22 examined a dependency boundary and a foreign-function boundary as single crossings, each demanding this book's admissibility discipline be either automatically enforced or manually reconstructed. This chapter examines what happens when several such crossings are chained in a single request's path, and argues, following the shipping company and the hospital, that the composed system's trustworthiness is not a new problem requiring new theory, but the same boundary discipline applied once per crossing, with the crossings' failures composing exactly as Chapter 2 described ordinary

functions composing.

24.2 General Principle: A Request Path Is a Composition of Boundary Crossings

Chapter 2 established that composing functions reliably requires each stage's declared state to actually capture what it depends on, and that a chain of such stages, $A \to B \to C$, is only as trustworthy as its weakest referentially transparent link. A networked request handler is, structurally, exactly this kind of chain: bytes arrive at a socket (A), are parsed into a typed value (B), and are used to query or update a store (C), with the response following the same chain in reverse. Each stage in this chain is a boundary in the sense Chapter 14 defined, a point where a claimed domain may exceed an honest domain, and each stage's failure mode is properly represented, in Chapter 14's sense, only if the `Result` or `Option` it returns carries evidence specific to that stage rather than collapsing every possible failure into an undifferentiated error.

The temptation particular to networked systems, and worth naming before the rest of this chapter's examples, is to treat the whole request path as a single unit and represent its failure with a single, coarse error type, exactly the evidence-erasure Chapter 14, Section 14.6 warned against, now multiplied across three or more boundaries instead of one. A response that says only "internal server error" for a malformed request body, a database connection failure, and a downstream service timeout has performed Chapter 8's exhaustiveness technically, some error was represented, while discarding exactly the boundary-specific evidence a caller, or an on-call engineer, needs to distinguish a client mistake from an infrastructure failure.

24.3 Rust Mechanism: Async as the Substrate, Not a New Set of Rules

A networked service's request handlers are typically written as `async fn`s, and Chapter 18 already supplies everything needed to reason about them correctly: a handler suspended at an `.await` point while waiting on a socket read or a database query is holding exactly the kind of reified continuation Chapter 18, Section 18.3 formalized, and any reference it captured across that suspension remains subject to the same extended-borrow discipline Chapter 18, Section 18.6 worked through. Nothing about network I/O specifically requires new theory; a request handler's correctness rests entirely on Chapters 5, 14, and 18 already established, applied to a specific, common shape of program.

```
async fn handle_request(pool: &DbPool, body: &[u8]) -> Result<Response,
    HandlerError> {
    let req: CreateOrder = serde_json::from_slice(body)
        .map_err(HandlerError::MalformedBody)?;          // boundary 1:
            wire format

    let order = insert_order(pool, &req).await
        .map_err(HandlerError::Storage)?;                 // boundary 2:
            data store

    Ok(Response::created(order))
}
```

24.4 Worked Example: Preserving Evidence Across a Chained Boundary

```
#[derive(Debug)]
enum HandlerError {
    MalformedBody(serde_json::Error),
    Storage(sqlx::Error),
}

impl HandlerError {
    fn status_code(&self) -> u16 {
        match self {
            HandlerError::MalformedBody(_) => 400, // client's fault
            HandlerError::Storage(_) => 502,        // this service's
                fault
        }
    }
}
```

This is Chapter 14, Section 14.5's `ConfigError` pattern, unchanged, applied to a request handler instead of a configuration loader: two distinct boundaries, each with its own failure mode, preserved in an enum rather than collapsed, and the preservation pays for itself immediately, since `status_code` can only distinguish a client error from a server error because the two boundaries' evidence was kept separate all the way through. A handler that collapsed both into a bare `String`, in the style of Chapter 14's `read_max_connections_v2`, could not implement `status_code` correctly at all; the information needed to choose between 400 and 502 would already be gone by the time this function ran.

24.5 Interpretation Exercise: A Chain That Looks Robust but Isn't

```
async fn handle_request_v2(pool: &DbPool, body: &[u8]) -> Response {
    let req: CreateOrder = serde_json::from_slice(body).unwrap_or_default
        ();
    match insert_order(pool, &req).await {
        Ok(order) => Response::created(order),
        Err(_) => Response::error(500),
    }
}
```

This compiles, never panics on a malformed body, and always returns some response. The question worth asking is: *which of this book's earlier interpretation exercises does `unwrap_or_default` here actually repeat?* It is Chapter 13, Section 13.6's exercise again: `unwrap_or_default` performs Chapter 8's exhaustiveness on `Result<CreateOrder, _>` while silently substituting a fabricated, empty `CreateOrder` for a malformed request, which then proceeds to `insert_order` as though the client had actually asked to create an empty order, an outcome the client never requested and the service never intended to honor. The failure has not been handled, it has been hidden one layer deeper, and it will surface later, if at all, as a confusing row in the order table rather than as the 400 response Section 24.4's version would have produced at the exact boundary where the real problem occurred.

24.6 Connections: Boundaries Compose the Same Way Functions Do

This chapter's central claim is deliberately modest: a networked service introduces no correctness problem this book has not already solved, individually, for a single boundary crossing. What it introduces is composition, in exactly Chapter 2's sense, of several such crossings in sequence, and the discipline required to keep a composed chain trustworthy is the same discipline Chapter 2 established for composing ordinary functions: each stage's declared state, here each stage's error type, must actually capture what that stage depends on and can fail at, or the composition's overall reliability silently degrades to that of its least carefully represented link. A reader who has internalized Chapters 2, 14, and 18 has already learned everything this chapter needed to teach; what this chapter has added is the observation that a request handler is where all three most commonly meet at once.

Invariant Established in This Chapter

A networked service's request path is a composition of boundary crossings, socket, wire format, data store, each individually answerable to the admissibility discipline this book has already developed for boundaries in general. The composed chain's trustworthiness is bounded by its least carefully represented crossing, not averaged across all of them; collapsing several boundaries' distinct failure modes into one undifferentiated error satisfies exhaustiveness in name while discarding exactly the evidence a caller needs to tell a client mistake from an infrastructure failure.

Chapter 25: Testing as Admissibility Verification

From First Principles

Question: If a type system can only verify what can be stated in its own vocabulary, what happens to every claim that falls outside it?

25.1 Phenomenon: Verification That Continues Where Proof Stops

A bridge's structural engineering is verified twice, in two different ways. First, on paper, calculations establish that the design, as specified, satisfies the relevant load equations; this is a proof, checked once, that holds for every bridge built correctly to that specification. Second, once built, the actual physical bridge is load-tested, weights driven across it, deflection measured, because the paper proof verified the *design*, not the specific concrete and steel that a particular construction crew actually poured and welded on a particular day. Neither form of verification replaces the other; the calculation proves something the load test cannot, that the design is sound for every instance built to it, and the load test proves something the calculation cannot, that this particular instance actually matches the design closely enough for the calculation's conclusions to apply to it.

A pharmaceutical compound's safety is established the same way, twice, in different registers. A chemist's structural analysis establishes, from first principles of molecular interaction, what the compound should do; a clinical trial then establishes, empirically, what it actually does in real bodies, because the structural analysis cannot fully anticipate every interaction a genuinely complex biological system might produce. This book has spent twenty-three chapters developing Rust's version of the paper calculation, exhaustively, formally, chapter after chapter of guarantees the compiler checks once and relies upon forever after. This final chapter asks what corresponds, in this book's own vocabulary, to the load test and the clinical trial: verification of exactly the claims the compiler's calculation could not make on its own.

25.2 General Principle: Testing Where Formal Verification's Vocabulary Runs Out

Chapters 15, 21, and 22 each identified a specific point where this book's formal machinery reaches its limit. Chapter 15 showed a panic as the discovery of a state the type system could

not rule out statically. Chapter 21 showed `unsafe` obligations that the compiler cannot verify and must instead be discharged by human argument. Chapter 22 showed an FFI boundary where no compiler on either side can catch a mismatch at all. In each case, the gap was not a defect in Rust's type system; it was a boundary of what any static type system, however expressive, can in principle check, because some claims depend on runtime values, external systems, or reasoning too intricate to express as a type.

A test is exactly the mechanism this book has been implicitly gesturing toward at each of these boundaries: an empirical check, executed against actual runtime behavior, of a claim the compiler either cannot state or cannot verify on its own. This reframes testing away from its common informal description, "making sure the code works," and toward something considerably more precise, given everything the preceding twenty-three chapters have built: a test verifies, for a specific, concrete case, exactly the kind of admissibility claim Chapters 5 through 23 spent their formal machinery trying to guarantee for every case at once. Where the compiler proves a property for an entire class of inputs, a test samples that property at one, or a deliberately chosen few, points, and reports whether the sample confirms or refutes it.

25.3 Formal Definition: A Test as a Point-Sampled Admissibility Claim

Recall Chapter 12's honest domain, $\mathrm{dom}(f) \subseteq A$, the inputs for which a function f genuinely behaves as intended. A unit test asserting a specific output for a specific input $a \in A$ is a claim that $a \in \mathrm{dom}(f)$ and that $f(a)$ equals the asserted value; passing confirms this for that one point, and says nothing, directly, about any other point in A. A property-based test, checking an assertion against many generated inputs, widens the sample without ever achieving the compiler's exhaustive, whole-domain guarantee; it remains a sampling of A, however large, rather than a proof over all of A.

This is why testing and the type-checking this book has spent twenty-three chapters developing are not competitors, and why neither can substitute for the other. A property the type system proves, exhaustiveness in a `match`, uniqueness of authority under ownership, holds for the entirety of Σ by construction, with no sampling involved at all; a property only a test can check, the correct numeric result of a specific business calculation, the actual behavior of a specific external system reached across an FFI boundary, holds, if it holds, only for the specific points actually tested, with every untested point remaining exactly as uncertain as it was before the test suite existed.

25.4 Rust Mechanism: The Test Harness as an Ordinary Consumer of the Type System

```
fn add_discount(price: f64, rate: f64) -> Option<f64> {
    if !(0.0..=1.0).contains(&rate) {
        return None;
    }
    Some(price * (1.0 - rate))
}

#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn rejects_rate_above_one() {
        assert_eq!(add_discount(100.0, 1.5), None);
    }

    #[test]
    fn applies_valid_rate() {
        assert_eq!(add_discount(100.0, 0.2), Some(80.0));
    }
}
```

Nothing about `#[test]` grants the test function any special relationship with Rust's type system; `rejects_rate_above_one` is an ordinary function, calling `add_discount` exactly as any other caller would, subject to every ownership and borrowing rule this book has developed. What the test harness contributes is orchestration, running many such ordinary functions and reporting which assertions held, not any new verification power beyond what Chapters 1 through 24 already gave every function in the program. This is worth stating because it clarifies exactly what Section 25.3 already implied: a test's assertion is verified with precisely the same compiler machinery as any other boolean expression in the language, its distinctive contribution lies entirely in *which* claim it chooses to check and *how many* points it samples, not in any deeper access to the program's semantics.

25.5 Worked Example: A Test Targeted at Exactly a Type-System Gap

```
#[test]
fn boundary_rate_of_exactly_one_is_admitted() {
    // rate == 1.0 is included by the inclusive range in
    // add_discount's guard; this test exists specifically
    // because that inclusivity is a choice the type system
    // itself has no vocabulary to state or verify -- Option<f64>
    // says nothing about which f64 values are excluded, only
    // that some are.
    assert_eq!(add_discount(100.0, 1.0), Some(0.0));
}
```

This test is not a generic sanity check; it is targeted precisely at the kind of gap Section 25.2 identified. `add_discount`'s signature, `fn add_discount(price: f64, rate: f64) -> Option<f64>`, states, honestly in Chapter 12's sense, that some inputs are rejected, but it says nothing about exactly which boundary, whether `rate == 1.0` is admitted or rejected, is a fact only the function's body decides and only a test, or a careful reading of that body, can confirm. This is the precise, narrow role testing plays throughout this book's framework: not a replacement for the honest-domain discipline Chapter 12 established, but a verification tool for the specific boundary decisions that discipline's types cannot themselves express.

25.6 Interpretation Exercise: A Passing Suite That Verifies Less Than It Appears To

```
#[test]
fn discount_is_reasonable() {
    let result = add_discount(100.0, 0.5);
    assert!(result.is_some());
}
```

This test passes. The question worth asking is: *what admissibility claim, in Section 25.3's sense, has this test actually sampled?* It confirms only that `add_discount(100.0, 0.5)` returns `Some(_)`, some value, unspecified; it does not confirm the value is `50.0`, or indeed anything about the actual arithmetic performed. A mutation to `add_discount`'s body, say, an accidental change from `price * (1.0 - rate)` to `price * rate`, would still return `Some(50.0)` for this particular input by coincidence, and this test would continue to pass, having verified nothing about the actual computation it appears, by its name, to be checking. This is the testing-era counterpart to Chapter 14's `map_err` exercise: a mechanism, the passing test, that has the correct outward shape of verification while silently failing to sample the specific claim, correct arithmetic, that actually mattered. A test's value is bounded by the specificity of what it asserts, exactly as a `Result`'s value in Chapter 14 was bounded by the specificity of what its error type preserved; the type system enforces that some assertion was checked, never that the assertion checked was the one that counted.

25.7 Connections: What the Whole Book Has Been Building Toward

This chapter closes a book that opened, in Chapter 1, by asking what a program actually is, and answered: a state transition, not a transcript. Every chapter since has been an elaboration of that single answer, ownership as the discipline governing who may transition a shared value, structs and enums as the shape of the states being transitioned between, traits and generics as the behavioral contracts governing transitions across type boundaries, `Option` and `Result` and panic as the honest representation of transitions that cannot always succeed, concurrency and async as that same discipline extended across simultaneously and non-continuously executing paths, and Cargo, macros, `unsafe`, FFI, and networking as the boundaries at which the discipline must be extended, suspended, manually rebuilt by hand, or composed across several crossings at once. Testing, this final chapter's subject, is not a coda tacked onto that structure; it is the acknowledgment that the structure, however thorough, was always going to have edges, points where a claim could be stated but not proven by the compiler alone, and that a disciplined engineer's response to such an edge is neither to pretend it does not exist nor to abandon rigor there, but to bring exactly the same care this book has modeled throughout, precise claims, honestly stated domains, exhaustively considered cases, to bear on verification by sampling rather than verification by proof. A reader who has internalized this book's vocabulary should, by this point, be able to look at any Rust codebase, their own or otherwise, and ask of any line: what admissible continuation is this claiming, who verified it, and by which of this book's two methods, compile-time proof or test-time sample, was that verification actually done.

It is worth closing on something this book's formal register has mostly left implicit: the

discipline described across these twenty-five chapters is demanding, and living inside it, day to day, is a genuinely different experience of programming than working in a language that lets ambiguity pass silently. Ellen Ullman's writing on a career spent inside the felt experience of exacting machines, the particular intimacy and the particular alienation of a discipline that will not let a vague thought pass as a finished one, and how that relationship shifts as both the technology and the person working with it change over years, captures something this book's technical chapters were never going to be the right place to say directly [8]. A language that forces every claim about ownership, every partial function, every boundary crossing to be stated honestly is asking something of the person writing it, not only of the compiler; the reward, a program whose guarantees can actually be trusted, is real, but so is the cost of the precision it demands, and a reader who has felt that cost across the exercises in this book has understood something about the discipline that no invariant box alone could have stated.

The next time something breaks, in a codebase or in any system this book's vocabulary has turned out to describe, it is worth reaching for a different first question than the usual one. Not *what broke*, but *who was holding the baton when it wasn't their turn*.

Invariant Established in This Chapter

Testing is admissibility verification by sampling rather than by proof: it checks a specific, concrete claim about runtime behavior at the exact points where this book's compile-time machinery, ownership, exhaustiveness, honest domains, cannot state or cannot verify the claim on its own. A test's value is bounded by the specificity of what it actually asserts, not by the fact that it passed; and the discipline this entire book has developed for stating types precisely, Chapter 6 through Chapter 24, applies with equal force to stating what a test actually checks, since a vague assertion samples an admissibility claim no more meaningfully than a dishonest type signature states one.

Part VI Summary

- A dependency graph is a trust graph, and semver is a voluntary, compiler-unverified admissibility contract over time (Chapter 19).
- A macro is a syntax-to-syntax function that earns no exemption from any guarantee this book has developed (Chapter 20).
- Unsafe code suspends automatic verification for a narrow, fixed set of operations; it never suspends the underlying obligation, which shifts to the programmer (Chapter 21).
- No guarantee crosses a foreign-function boundary automatically; every one of them must be manually reconstructed at the point of crossing (Chapter 22).
- Performance and correctness usually trace back to the same underlying property, locality, viewed through two lenses (Chapter 23).
- A networked request path composes several boundary crossings, socket, wire format, data store, each individually answerable to this book's admissibility discipline (Chapter 24).
- Testing verifies, by sampling, exactly the admissibility claims the compiler's proof could not state or could not check on its own (Chapter 25).

Therefore: every boundary a Rust program eventually crosses, to other people's code, to generated code, to unchecked operations, to another language entirely, to physical hardware, to a chain of networked services, to the future behavior no static check can fully anticipate, remains answerable to the single admissibility discipline developed across Parts I through V. Only the mechanism of enforcement changes, from automatic and exhaustive to explicit, sampled, or rebuilt by hand; the obligation itself never does.

Appendix A

These Ideas in Practice: A Tour of the Author's Own Rust Projects

The vocabulary this book has developed, admissibility, reachability, continuation, locality, was not invented for the purpose of writing about Rust. It is the same vocabulary this author has used, across a separate and ongoing body of work, to reason about constraint, repair, and legitimate state transition in systems well outside programming. This appendix is a short tour of a few of that work's Rust artifacts, read back through the lens this book has built, not as case studies proving the book's claims, but as evidence that the vocabulary was doing real work before Chapter 1 ever mentioned Rust.

A.1 physiome: Ownership as a Homeostatic Discipline

`physiome` is a whole-body physiology simulator, in the tradition of the HumMod model, built around a pure functional core: immutable state, explicit effects, with homeostatic correction governed by admissibility and repair theory rather than by a fixed system of differential equations alone. Concretely, every physiological variable carries an admissible boundary, a range the rest of the organism will accept, and a departure from that boundary triggers the dispatch of a declared repair operator against it, rather than deriving a correction from whatever equation produced the departure in the first place; the project states this as encoding *Boundary* $\rightarrow$ *Violation* $\rightarrow$ *Repair* $\rightarrow$ *Behaviour* in place of the more familiar *Mechanism* $\rightarrow$ *Behaviour*. Read against Chapter 5, the project's central design commitment, immutability by default, is a direct application of that chapter's central claim: a value's admissible continuations are only as trustworthy as the guarantee that no untracked path can silently redirect them, and every repair operator and subsystem advance in the implementation takes the whole-organism state by immutable reference and returns a new state by value, with no interior mutability anywhere in its subsystem modules. A physiological simulation with dozens of interacting organ systems is exactly the setting where Chapter 3's cost of global knowledge becomes acute, since a naive, freely-mutable shared state would force every subsystem's update to be reasoned about in light of every other subsystem's potential interference. Structuring the simulator's core around explicit, owned state transitions, rather than shared mutable structures, is Chapter 4's locality principle applied at the scale of an entire organism model: each organ system's update function can be reasoned about, and tested, using only its own declared inputs and out-

puts, exactly the bounded reasoning footprint Chapter 4 formalized as $N(f, v)$; the project's own account of a defect where a single directionally-backwards repair operator drove a variable to an absurd extreme is a real instance of exactly the locally-checkable failure Chapter 4's framework predicts such an architecture should produce, one operator's contract violated in isolation, rather than a defect requiring the whole system to be held in mind to diagnose.

A.2 Spherepop: Composition Over an Inherited Monolith

Spherepop, a programming language and formal calculus developed across a long research program, arrived, after several rounds of revision, at four primitive operators, Pop, Refuse, Bind, and Collapse, corresponding to selection, exclusion, coupling, and projection, with every other construct in the language, Sphere, Merge, Choice, Link, Unlink, SetMeta, treated as second-order syntactic sugar expressible in terms of these four. This is Chapter 11's argument, composition over inheritance, playing out at the level of an entire language's design rather than a single Rust type: rather than a single monolithic calculus offering every operation a user might conceivably need, the mature Spherepop calculus offers a minimal, composable basis, analogous to functional completeness in Boolean logic, from which every richer construct is derived rather than separately special-cased. The Rust reference implementation of this kernel, organized into event, history, collapse, arbiter, overlay, and sugar modules, mirrors Chapter 9's admissibility-contract framing directly: the sugar module depends on the kernel modules only through their declared interfaces, never their internals, which is precisely what lets the surface calculus be revised, as it has been more than once across the project's history, without the underlying four-primitive kernel needing to change.

A.3 entheai's RepairLedger: Panic Formalized as Data

The `entheai` project's `repair_stop.rs` implements a `RepairLedger` governed by a `StopReason` enum, currently distinguishing three ways a repair process can conclude: `MaxAttempts`, `NoMarginalValue`, and `WeakVerifier`. Read against Chapter 15, this is a genuinely different, and arguably more disciplined, response to the same problem Chapter 15 examined: rather than letting a departure from an admissible manifold surface as an unstructured panic, `RepairLedger` reifies the departure as an ordinary, inspectable value, exactly the move Chapter 13 described for turning an unrepresented possibility into a named one, applied here not to absence but to failure-to-continue-repairing. `StopReason`'s three variants are, in this book's vocabulary, three distinct ways a repair computation's honest domain, in Chapter 12's sense, can be exceeded, each named and handled explicitly rather than collapsed into a single undifferentiated halt. The project's own open question at time of writing, whether `StopReason` should be widened to represent a full cost-weighted objective rather than a single scalar margin, is itself a live instance of Chapter 14's concern: whether the current representation propagates enough evidence about *why* a repair stopped, or merely records *that* it did.

A.4 Attestal: Provenance as a Boundary Contract

Attestal, an illustrative Rust program built around a deliberately named function `dangerously_compress()`, demonstrates how compressing away a result's provenance creates room for a fabricated his-

tory to be attributed to it later. Read against Chapter 19's treatment of trust graphs, the underlying claim, that provenance is not metadata about an artifact but the retained constraint on what histories may subsequently be attributed to it, is the same shape as this book's treatment of a dependency's semver promise: both are claims that something crossing a boundary, a compressed result, a published crate version, carries with it a constraint on what can legitimately be said about it afterward, and both fail in the same way when that constraint is silently discarded rather than checked. A crate whose actual behavior diverges from what its version number promised, examined in Chapter 19's worked example, and an artifact whose actual history diverges from what its provenance record promised, examined in Attestal, are two instances of a single underlying failure: a boundary crossing that looked, from the outside, like it preserved a guarantee it had actually already lost.

A.5 A Closing Observation

None of these four projects were built with this book in mind; the book came later, applying a vocabulary these projects, and a substantial body of writing alongside them, had already been developing for other purposes. That the vocabulary transfers this cleanly onto Rust specifically, rather than requiring strained analogy, is the closest thing this appendix offers to independent evidence for the book's central claim: that Rust's own design was, whether or not its authors used this language for it, already an instance of the same admissibility discipline this author has spent a great deal of other work trying to state precisely.

Appendix B

Quick Reference: The Rules, Compressed

The preface promised this appendix: the same short list of Rust rules that circulates informally, restated here in compressed form now that every one of them has actually been derived, with a pointer back to the chapter that did the deriving and, where the informal version overstates or skips a step, a note correcting it.

One owner. When the owner's scope ends, the value's resources are released.

This is Chapter 5's central claim, that a value's continuation has exactly one author at a time, restated as a rule of thumb. The compressed version is accurate as far as it goes, but it is worth remembering what it is shorthand for: not merely a deallocation timer, but a guarantee that responsibility for what happens to a value next is never diffused across more than one path. "No garbage collector needed" is a true and useful consequence, not the reason the rule exists; a garbage collector solves the deallocation half of the problem Chapter 5 Section 5.1 describes and leaves the aliasing half untouched, which is why ownership is a strictly stronger guarantee than collection, not merely a faster one.

Share for reading, borrow exclusively for writing, never both at once.

This is Chapter 5, Sections 5.4 and 5.5. The informal justification usually offered, that this "stops data chaos," names an outcome without naming a mechanism. The actual mechanism, derived across Chapters 3 through 5, is that mutation invalidates the beliefs of any other path that reached the value first (Chapter 3), and that the cost of checking whether such an invalidation is safe scales with how many paths can reach the value (Chapter 5, Section 5.1's reachability set $R(v)$). Rust's aliasing rule does not check this cost at every mutation; it eliminates the cost by guaranteeing, structurally, that no other path can be relying on stale beliefs at the moment a mutation occurs. "Stops chaos" is the effect; "removes the possibility of another path holding a now-false belief" is the mechanism, and only the second one tells you what to do when the compiler rejects code that looks, to you, perfectly safe.

No null. A possibly-missing value must be represented and handled.

This is Chapters 12 through 14. The informal version is right that `Option` and `Result` force the missing or failed case to be handled before a value can be used, but it undersells what is actually happening: Chapter 12 shows that almost every partial function has a claimed domain wider than its honest domain, and Chapters 13 and 14 show that `Option` and `Result` are not error-handling conveniences bolted onto the language, they are the mechanism by which a function's type signature is made to tell the truth about its own honest domain. A function returning a bare `T` where absence or failure is actually possible has not avoided the problem; it has hidden it, moving the discovery from compile time to whatever runtime moment first exercises the untested path (Chapter 15).

Plan data flow before writing code: who creates it, who reads it, who cleans it up.

This is good practical advice, and it corresponds to Chapter 2's composition argument and Chapter 4's locality argument taken together: a design where every function's actual dependencies match its declared signature (Chapter 2) and where each component's correctness can be verified from a bounded neighborhood (Chapter 4) is, in practice, a design where data flow was planned rather than discovered by accident. The advice is a design heuristic; Chapters 2 and 4 are the reason the heuristic works.

Treat the compiler's objections as a description of a design flaw, not an obstacle.

This is Chapter 5, Section 5.7: the borrow checker is a static verifier for a specific, checkable claim, not a linter enforcing a style preference. When it rejects a program, in the vocabulary this book has built throughout, it has found a reachability path the program's structure permits but which the program's logic did not actually account for. Chapters 15, 20, 21, and 22 extend this same idea to panics, macros, unsafe code, and FFI boundaries respectively: in every case, an error or an obligation surfacing at one of these edges is evidence about the design, not merely friction to be routed around.

Make illegal states unrepresentable.

This is Chapters 6 and 7 together: a struct's fields fix a product state space (Chapter 6), an enum's variants fix a closed sum (Chapter 7), and a value's admissible state space is exactly what its declared type specifies, no more and no less. "Illegal states unrepresentable" is what it looks like from the outside when a design has actually used this machinery deliberately, narrowing a type's fields and variants until the type itself cannot express anything the domain considers invalid, rather than relying on a comment, a runtime check buried somewhere in the logic, or a convention the next contributor might not know about.

A Closing Note on Compression

Every rule in this appendix reads as a single sentence because a single sentence is what a reader wants to carry around day to day, glanced at before writing a function rather than re-derived from Chapter 1 each time. That is a legitimate and useful thing for a rule to be. What this appendix has tried to preserve, in each entry, is a pointer back to the chapter where

the sentence stopped being a slogan and became a conclusion, because the moment a rule is needed outside the case it was memorized for, having the argument, not just the sentence, is the difference between adapting the rule correctly and merely repeating it.

Appendix C

Appendix C: A Practical Companion

This book's chapters derive Rust's rules before showing its syntax, which is the right order for building durable understanding but a deliberately slow order for a reader who simply needs to know how to write a test, publish a crate, or parse a command-line flag this afternoon. This appendix inverts the order for exactly four practical topics, syntax and mechanics first, each section closing with a pointer back to the chapter that actually derived the underlying principle, so that the practical shortcut never loses its connection to the argument behind it.

C.1 Testing Mechanics

A test function is any function annotated `#[test]`, typically gathered in a `mod tests` block guarded by `#[cfg(test)]` so the module compiles only when running tests. The core assertion macros are `assert!(condition)`, `assert_eq!(left, right)`, and `assert_ne!(left, right)`, each panicking with a diagnostic message on failure. A test expected to panic is annotated `#[should_panic]`, optionally with an `expected = "substring"` argument narrowing which panic message is acceptable. `cargo test` runs every test in the crate by default; a substring argument, `cargo test discount`, runs only tests whose name contains it, and `cargo test -- --nocapture` shows output from passing tests as well as failing ones.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn adds_positive_numbers() {
        assert_eq!(add(2, 3), 5);
    }

    #[test]
    #[should_panic(expected = "divide by zero")]
    fn rejects_zero_divisor() {
        divide(1, 0);
    }
}
```

A common workflow, test-driven development, writes the `#[test]` function and its assertions before the function under test exists, then implements just enough to make the test pass. Nothing about this workflow is specific to Rust, but Rust's ownership and borrowing rules apply to test code exactly as they apply anywhere else, so a test that would not compile under the borrow checker is telling you the same thing any other rejected code tells you. *See Chapter 25 for why testing is best understood as admissibility verification by sampling, and Chapter 8 for why an assertion's exhaustiveness is not the same as its correctness.*

C.2 Cargo and Crates.io Workflow

Publishing a crate begins with the metadata in `Cargo.toml`: `name`, `version`, `edition`, and, for a public crate, `description`, `license`, and `repository`. `cargo login` stores an API token from crates.io; `cargo publish` then builds, packages, and uploads the crate, after `cargo package --list` has confirmed exactly which files will be included. Versioning follows the semantic-versioning discipline this book examines at length: a patch release, `0.1.1`, should change nothing about the public API; a minor release, `0.2.0`, may add but not remove or change existing public items; a major release, `1.0.0` to `2.0.0`, may break compatibility outright, and `cargo semver-checks` (a separate, widely used tool) can catch accidental breakage before a release that was only meant to be minor.

A workspace, declared with a top-level `[workspace]` table listing member crates, lets several related crates share a single `Cargo.lock` and build directory, useful once a project's binary, library, and test-support code are cleanly separated into their own crates rather than crammed into one. A custom subcommand, any executable named `cargo-foo` placed on the `PATH`, becomes callable as `cargo foo`, which is how tools like `cargo-watch` and `cargo-audit` integrate into the ordinary workflow without modifying Cargo itself. *See Chapter 19 for why a dependency graph is best understood as a trust graph, and why a committed lockfile pins less than it might seem to.*

C.3 Macro Authoring Mechanics

A `macro_rules!` definition matches one or more patterns against the tokens supplied at a call site; each pattern captures fragments using a designator, `$name:expr` for an expression, `$name:ident` for an identifier, `$name:ty` for a type, among others, and repetition is expressed with `$(...)*` or `$(...),+`, matching zero-or-more or one-or-more repetitions respectively, optionally separated by a literal token.

```
macro_rules! max_of {
    ($first:expr) => { $first };
    ($first:expr, $($rest:expr),+) => {
        {
            let rest_max = max_of!($($rest),+);
            if $first > rest_max { $first } else { rest_max }
        }
    };
}

let m = max_of!(3, 7, 2, 9, 4); // expands recursively to 9
```

A derive macro, written as a separate procedural-macro crate exposing a function annotated `#[proc_macro_derive(Name)]`, generates an `impl` block from a struct or enum's definition at compile time; this is how `#[derive(Debug)]`, `#[derive(Clone)]`, and `serde`'s `#[derive(Serialize, Deserialize)]`, used in Chapter 14's worked example, are implemented. Debugging a stubborn macro expansion is usually best done with `cargo expand` (a separate, commonly installed tool), which prints the fully expanded code a given invocation actually produces, turning an opaque compiler error inside a macro's output into ordinary code that can be read and debugged directly. *See Chapter 20 for why a macro is best understood as a function from syntax to syntax, earning no exemption from any guarantee derived elsewhere in this book, and why hygiene is Chapter 5's reachability discipline extended across the expansion boundary.*

C.4 General-Purpose Recipes

A handful of small, frequently needed tasks are worth having in one place rather than rediscovering separately each time they come up.

Command-line arguments. `std::env::args()` yields the raw argument strings; for anything beyond the simplest case, the `clap` crate's derive API turns a struct into a full parser:

```
#[derive(clap::Parser)]
struct Cli {
    #[arg(short, long)]
    verbose: bool,
    input: String,
}

let cli = Cli::parse();
```

File I/O. `std::fs::read_to_string(path)` reads an entire file into a `String`, returning a `Result` exactly in Chapter 14's sense; `std::fs::write(path, contents)` writes one, creating or truncating the file as needed.

Regular expressions. The `regex` crate compiles a pattern once, ideally cached behind a `once_cell` or `std::sync::LazyLock`, and reuses it across calls to `is_match`, `find`, or `captures`.

Logging. The `log` crate's macros, `info!`, `warn!`, `error!`, provide a stable logging interface independent of which backend actually prints or ships the messages; a backend crate such as `env_logger` or `tracing-subscriber` is initialized once, typically as the first line of `main`, to actually consume them.

Subprocesses. `std::process::Command::new("program").arg("value").output()` spawns a child process and collects its output, itself a boundary crossing in this book's sense, since the child process's exit status and output are exactly the kind of partial, honestly-typed result Chapter 12 argues every boundary should return rather than assume.

None of these four recipes required anything beyond what earlier chapters already derived; each is simply a common enough task to be worth writing down once. *See Chapter 2 for the composition principle every one of these small utilities ultimately rests on, and Chapter 12 for why so many of their return types are `Result` or `Option` rather than a bare value.*

Appendix D: References

This book is, by design, mostly self-contained: its central arguments are derived from first principles rather than assembled from citations. A small number of ideas from outside programming, however, are close enough to specific chapters' arguments that naming them directly, rather than leaving the resemblance implicit, seemed the more honest choice. The four works below are cited at the point in the text where each resemblance is closest.

Bibliography

[1] Mark Wilson. *Wandering Significance: An Essay on Conceptual Behaviour*. Oxford University Press, 2006.
Cited in Chapter 4, on the patchwork, locally-valid structure of physical and mathematical concepts as a precedent for treating a function's correctness as a locally verifiable claim rather than a globally presupposed one.

[2] Alicia Juarrero. *Dynamics in Action: Intentional Behavior as a Complex System*. MIT Press, 1999.
Cited in Chapter 5, on context-dependent constraint as generative rather than merely restrictive, as a precedent for reading Rust's ownership rule as an enabling condition for safe concurrency rather than only a restriction against it.

[3] Karl M. Fant. *Logically Determined Design: Clockless System Design with NULL Convention Logic*. Wiley, 2005.
Cited in Chapter 16, on clockless, delay-insensitive hardware design as a coordination discipline with no single timekeeper, the hardware-level counterpart to this chapter's channels and mutexes: correctness follows from local handshakes being honored rather than from a shared clock signal.

[4] Catherine Liu. *Virtue Hoarders: The Case Against the Professional Managerial Class*. University of Minnesota Press, 2021.
Cited in Chapter 14, on institutional abstraction that formally accounts for a decision while discarding the evidence needed to trace its consequences back to whoever made it, as a social-scale parallel to this chapter's argument against collapsing a boundary's distinct failure modes into an undifferentiated error.

[5] Sarah Wynn-Williams. *Careless People: A Cautionary Tale of Power, Greed, and Lost Idealism*. Flatiron Books, 2025.
Cited in Chapter 14, alongside Liu, on early organizational decisions abstracted away from their eventual human consequences, propagating outward with the evidence of who decided what already stripped from them by the time they reach whoever bears the cost.

[6] Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press, 2005.
Cited in Appendix E, on building an entire computer, gate by gate and layer by layer from NAND alone, as a full-length precedent for that appendix's own miniature exercise in deriving a complete gate family from a single operator and verifying each layer only against the one beneath it.

[7] Douglas Hofstadter and Emmanuel Sander. *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking*. Basic Books, 2013.
Cited in the Preface, on analogy-making as close to the core mechanism of thought rather than a peripheral aid, as the working premise behind this book's heavy reliance on non-programming analogies to introduce each chapter's technical content.

[8] Ellen Ullman. *Life in Code: A Personal History of Technology*. MCD/Farrar, Straus and Giroux, 2017.
Cited in Chapter 25, on the felt, personal experience of working closely with a machine that demands precision, and how that experience changes across a career spent inside it, as a closing acknowledgment of what this book's technical chapters ask of the person writing the code, not only of the compiler checking it.

[9] Bartosz Milewski. *Category Theory for Programmers*. Self-published, 2019.
Cited throughout Chapters 1, 2, 6, 7, 10, 13, 14, and 18, and in Appendix F, as the standard reference for the categorical vocabulary, morphism, product, coproduct, functor, monad, this book deliberately withholds until each corresponding idea has already been derived from concrete systems reasoning.

[10] Erik Meijer. "The Curse of the Excluded Middle." *Communications of the ACM*, 57(6), 2014.
Cited in Chapter 13, on the practical cost of forcing a false choice between treating all computation as total and abandoning the tracking of partiality altogether, as a precedent for `Option<T>` as a middle path that represents partiality honestly rather than avoiding it.

[11] Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010.
Cited in Chapter 23, on sparsity as a single underlying property that simultaneously yields storage efficiency and reconstructibility, as a parallel to locality's simultaneous yield of human-verifiable correctness and cache-friendly performance.

Appendix D

Appendix E: Logic Gates from Spherepop-Style Operators

Appendix A mentioned Spherepop, a separate research program of the author's, as an instance of Chapter 11's composition-over-inheritance argument: its mature calculus reduces every construct to four primitive operators, **Pop** (selection), **Refuse** (exclusion), **Bind** (coupling), and **Collapse** (projection), with everything else built from them rather than separately special-cased. This appendix makes that claim concrete in Rust, by building ordinary Boolean logic gates directly on top of Rust analogues of those four primitives, and using the result to revisit Chapter 7's enum machinery, Chapter 9's trait contracts, and Chapter 16's citation of Karl Fant's clockless logic design from a single small, worked example.

E.1 The Four Primitives as a Two-Valued Enum

Spherepop's primitives are defined generally, over arbitrary admissibility structures, not merely over Boolean values; the version built here specializes them to the simplest possible case, a two-valued `Signal`, exactly the way Chapter 7 described an enum's variants as a closed, exhaustively-checked sum.

```
#[derive(Clone, Copy, PartialEq, Debug)]
enum Signal {
    Admit,
    Refuse,
}

use Signal::{Admit, Refuse as Excluded};
```

The name `Admit` rather than `true`, and `Refuse` rather than `false`, is deliberate: it keeps the connection to this book's admissibility vocabulary visible in the code itself, rather than translating it away into a bare Boolean the moment an example needs one.

E.2 Pop, Refuse, Bind, and Collapse

```
fn pop(a: Signal) -> Signal {
```

```
    a // selection: a single signal's own admissibility, unchanged
}

fn refuse(a: Signal) -> Signal {
    match a {
        Admit => Excluded,
        Excluded => Admit,
    } // exclusion: the complement of a signal's admissibility
}

fn bind(a: Signal, b: Signal) -> Signal {
    match (a, b) {
        (Admit, Admit) => Admit,
        _ => Excluded,
    } // coupling: joint admissibility, both signals admitted at once
}

fn collapse(a: Signal, b: Signal) -> Signal {
    refuse(bind(refuse(a), refuse(b))) // projection: admissible if either
        is
}
```

`bind` is ordinary conjunction, and `collapse` is defined, deliberately, not as a fourth independent case analysis but as `refuse` composed with `bind` composed with `refuse` on each input, the same De Morgan duality Spherepop's own calculus uses to derive projection from coupling and exclusion rather than treating it as a fifth primitive. This is Chapter 9's admissibility-contract discipline in miniature: `collapse`'s caller depends only on its declared behavior, admissible if either input is, and nothing about that caller needs to know, or could tell from the type signature alone, that the implementation is secretly reusing `bind` and `refuse` rather than pattern-matching independently.

E.3 NOR as the Single Generating Operator

Spherepop's own corpus establishes, as a proved theorem, that NOR alone is functionally complete: every other gate is expressible in terms of it, the same completeness property Boolean logic's NOR and NAND gates are each independently known to have. Building NOR from `collapse` and `refuse` directly, then deriving every other gate from NOR alone, demonstrates the same completeness claim as executable Rust rather than as a theorem statement.

```
fn nor(a: Signal, b: Signal) -> Signal {
    refuse(collapse(a, b))
}

fn not(a: Signal) -> Signal {
    nor(a, a)
}

fn or(a: Signal, b: Signal) -> Signal {
    not(nor(a, b))
}
```

```
fn and(a: Signal, b: Signal) -> Signal {
    nor(not(a), not(b))
}

fn xor(a: Signal, b: Signal) -> Signal {
    let d = or(a, b);
    let c = nor(a, b);
    and(d, not(c))
}
```

Not one of `not`, `or`, `and`, or `xor` pattern-matches on `Signal` directly; every one of them is built by composing `nor` with itself and with the gates already derived from it, exactly Chapter 11's claim that a rich vocabulary of capabilities can be composed from a minimal, independently verified basis rather than each requiring its own from-scratch implementation.

E.4 Verifying the Composition, Not Just Trusting It

Appendix C's testing mechanics apply here directly: each derived gate's truth table can be checked exhaustively, all four input combinations, against the ordinary Boolean definition, giving Chapter 25's admissibility-verification-by-sampling something concrete to sample.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn and_matches_truth_table() {
        assert_eq!(and(Admit, Admit), Admit);
        assert_eq!(and(Admit, Excluded), Excluded);
        assert_eq!(and(Excluded, Admit), Excluded);
        assert_eq!(and(Excluded, Excluded), Excluded);
    }

    #[test]
    fn xor_matches_truth_table() {
        assert_eq!(xor(Admit, Admit), Excluded);
        assert_eq!(xor(Admit, Excluded), Admit);
        assert_eq!(xor(Excluded, Admit), Admit);
        assert_eq!(xor(Excluded, Excluded), Excluded);
    }
}
```

Because every gate above is exhaustive over a two-variant enum, Chapter 8's reconstruction guarantee applies to each `match` directly: there is no third `Signal` variant a future change could introduce silently, and any attempt to add one would immediately break every non-exhaustive `match` in this module at compile time, exactly the safeguard Chapter 7 built the whole enum chapter around.

E.5 The Connection to Clockless Logic

Chapter 16 cited Karl Fant's Null Convention Logic as a hardware-level instance of coordination without a global clock, each gate's completion signaled locally rather than by a shared timing signal. NCL implements exactly the gates built here, physically, using dual-rail encoding in which a wire pair's four possible states represent `Admit`, `Refuse`, an intermediate not-yet-arrived state, and an unused fourth combination reserved as invalid, so that a gate's own inputs, not a clock edge, tell every downstream gate when a valid result has actually arrived. The `Signal` enum built in this appendix is a simplification of that four-state design down to Rust's ordinary, clocked execution model, but the underlying idea, that a logic gate's correctness should follow from its inputs' own admissibility rather than from an externally imposed timing assumption, is the same idea Chapters 5 through 25 have been calling admissibility throughout, now traced all the way down to the hardware that eventually executes the compiled program.

E.6 A Smaller Instance of a Larger Pedagogical Project

This appendix's method, deriving a complete family of logic gates from a single minimal operator, then treating everything built afterward as composed rather than separately implemented, is a small-scale instance of the project Noam Nisan and Shimon Schocken carry out at full length: building an entire modern computer, gates, chips, an assembler, a virtual machine, a compiler, and an operating system, one layer at a time, from nothing but NAND gates at the foundation [6]. Their project and this appendix share a working assumption worth stating plainly: that a system's trustworthiness is easiest to establish when each layer's correctness can be verified using only the layer beneath it, never by re-examining every gate simultaneously from the transistors up. This is Chapter 4's locality principle again, now applied not to a single Rust function but to the entire vertical stack a program ultimately runs on, evidence that the argument this book has made about software holds just as well one level down, in the hardware software depends on.

Appendix E

Appendix F: Correspondence with Category Theory

This appendix introduces no new content and changes no earlier chapter's argument. Its purpose is narrower: readers who already know category theory, or who go on to learn it after finishing this book, will notice that many of the constructions derived here from Rust and systems reasoning have standard names in a much older and more general mathematical language. This appendix names that correspondence directly, without requiring it for anything the book has already argued. The main text deliberately withheld this vocabulary until each idea had been derived on its own terms; a handful of "Connection to Category Theory" sidebars, in Chapters 1, 2, 6, 7, 10, 13, 14, and 18, made the same correspondence in passing, at the point each concept was introduced, and this appendix simply collects them in one place.

This book's term	Category-theoretic correspondence
State transition (Chapter 1)	Morphism
Composition of transitions (Chapter 2)	Composition
A transition that changes nothing (Chapter 2)	Identity morphism
Struct, product state space (Chapter 6)	Product
Enum, disjoint-sum state space (Chapter 7)	Coproduct
Pattern matching's case analysis (Chapter 8)	Case elimination on a coproduct
Trait as behavioral contract (Chapter 9)	Interface; loosely, a category of types sharing a signature
Generic bound satisfied by every instance (Chapter 10)	Parametric polymorphism
A type with a lawful `map` (Chapter 10)	Functor
`Option<T>` (Chapter 13)	Maybe (functor and monad)
`Result<T, E>` (Chapter 14)	Either (functor and monad)
The `?` operator's propagation (Chapter 14)	Kleisli composition
Iterator's `fold` (Chapter 10, Appendix C)	Catamorphism
`async`/`.await` sequencing (Chapter 18)	Monadic sequencing (an advanced correspondence, not required to follow Chapter 18)
An invariant preserved across a state transition (throughout)	A property preserved under a morphism

None of the correspondences above should be read as an exact, checked isomorphism. Rust's `Option` and `Result` satisfy the functor and monad laws by convention, verified informally the way Chapter 10's functor law was derived rather than checked automatically by the compiler the way a type class's laws might be verified in a language with that mechanism; a Rust type can implement `map` in a way that type-checks while quietly violating the law Chapter 10 stated, and nothing in the language itself will catch it. The correspondence in this table is instructive, a way of connecting this book's vocabulary to a much larger existing literature, not a claim that Rust enforces categorical laws the way it enforces ownership.

Readers who want to see this vocabulary developed on its own terms, starting from morphisms and composition and building up through functors, natural transformations, and monads to considerably more general structures than this book ever needed, are pointed to Bartosz Milewski's *Category Theory for Programmers* [9], cited throughout the sidebars above. That book and this one are, in a sense, traveling in opposite directions toward a shared middle: this one starts from a concrete language and its concrete failures and arrives, occasionally, at a general pattern; Milewski's starts from the general pattern and arrives, throughout, at concrete code. A reader who has finished *Thinking Like Rust* already has working intuition for several of category theory's central ideas, morphism composition, products and coproducts, functors,

monads, without necessarily knowing they were category theory at all; this appendix's only job was to say so.