"Type systems are the parts of formal methods that we've figured out how to make easy."

— Graydon Hoare (paraphrased)

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 9 of 10

Rust as a Proof Assistant

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Section 1

Introduction

What This Lecture Is About

Core claim: Rust's type system is, in a precise mathematical sense, a theorem prover embedded in a systems language.

Part I — The landscape

- What verification tools exist and why
- Where Lean 4 fits among them

Part II — Rust for Lean programmers

- Structs, enums, traits, ownership
- Idiomatic Rust patterns

Part III — The theory

- Substructural logics
- Ownership = affine types
- Borrowing = region-safe aliasing
- Traits = proof obligations

Part IV — Formal verification of Rust

- RustBelt, Aeneas, Prusti
- Rust programs verified in Lean 4

By the end, compiling Rust and type-checking a Lean proof should feel like the same activity.

Section 2

The Verification Landscape

Why So Many Proof Assistants?

Each tool makes different trade-offs across five axes:

1. **Expressiveness**: How much can the type system say? Dependent types? HoTT? Refinement types?
2. **Automation**: How much can the tool discharge automatically? SMT? External ATPs? Tactics?
3. **Trusted computing base**: How small is the verified kernel? (The De Bruijn criterion)
4. **Target domain**: Mathematics? Software? Hardware? Cryptography?
5. **Extraction**: Can a checked proof become running code?

No tool wins on all axes simultaneously. Knowing the landscape lets you choose the right tool.

Coq / Rocq — Features

Foundation: Calculus of Inductive Constructions (CIC)

Logic: Intuitionistic; propositions live in `Prop` (proof-irrelevant)

Proof style: tactic scripts (`intros`, `induction`, `rewrite`…)

Automation: `omega`, `ring`, `auto`, `eauto`, `tauto`

Extraction: proofs → OCaml / Haskell / Scheme

```
Theorem plus_comm :
  forall n m : nat,
    n + m = m + n.
Proof.
  intros n m.
  induction n.
  - simpl.
    rewrite <- plus_n_O.
    reflexivity.
  - simpl.
    rewrite IHn.
    rewrite <- plus_n_Sm.
    reflexivity.
Qed.
```

Good for: Verified compilation, large mathematical formalization, working with Iris / separation logic

Coq / Rocq — Notable Work

CompCert (Leroy, 2006–present)

- Verified C compiler: correct by construction
- No miscompilation bugs in the verified passes
- Used in avionics (DO-178C) and automotive

Four Colour Theorem (Gonthier, 2005)

- First fully mechanised proof
- Previously controversial (relies on computer search)
- The Coq proof eliminates the doubt

Feit–Thompson (Odd Order) (2012)

- 150,000 lines of Coq
- Original paper proof was 255 pages
- Cornerstone of finite group theory

Iris (Jung et al., 2015–present)

- Higher-order separation logic framework
- Built on top of Coq
- Used to verify Rust's type system (RustBelt)
- We return to this later today

Agda — Features

Foundation: Martin-Löf Type Theory (intensional)

Logic: Full dependent types; propositions *are* types, all the way down

Proof style: proof *terms* — no tactics, proofs look like programs

Syntax: Unicode-rich, flexible mixfix operators

If you know Lean 4, Agda is immediately readable.

```
-- Length is tracked in the type.
data Vec (A : Set) :
         Nat -> Set where
  []  : Vec A zero
  _::_ : {n : Nat}
       -> A
       -> Vec A n
       -> Vec A (suc n)

-- Safe head: the nil case
-- is impossible by type.
-- No partiality anywhere.
head : {A : Set}{n : Nat}
     -> Vec A (suc n) -> A
head (x :: _) = x
```

Good for: HoTT / Cubical type theory, categorical semantics and PL metatheory, maximum expressiveness with no automation

F* — Features

Foundation: Dependent types + **effects** + **refinement types** + SMT (Z3)

Logic: Intuitionistic with a classical SMT fragment

Automation: Z3 discharges arithmetic and memory-safety goals automatically

Extraction: F* $\longrightarrow$ C / OCaml / WASM

```
(* Refinement type: nonzero *)
val div :
    int ->
    x:int{x <> 0} ->
    int
let div a b = a / b

(* Z3 proves this automatically:
   no proof script needed *)
let add_comm (a b : nat) :
    Lemma (a + b = b + a) = ()

(* Effect tracking in the type *)
val readFile :
    string ->
    ML string
```

Good for: Verified cryptography, systems (with I/O, state, exceptions), programs where the spec is an arithmetic constraint

F* — HACL* and Verified Cryptography

Project Everest (MSR / INRIA / CMU): the first verified, production-grade TLS stack.

HACL* (High-Assurance Crypto Library)

- ChaCha20, Poly1305, Curve25519, BLAKE2, SHA-2, HMAC, Ed25519
- Everything proven correct and const-time (no timing side-channels)
- Extracted to C and shipped in **Firefox**, **Linux**, WireGuard

miTLS*

- Verified implementation of TLS 1.3
- Interoperates with real browsers

Dependent types and effect tracking are not just a theoretical exercise. HACL* is running in your browser right now.

Toolchains exist that translate Rust code to F* automatically. The workflow is:

1. Write in safe, "idiomatic" Rust
2. Extract an F* model automatically
3. Verify using HACL*-style techniques
4. Ship with an auditable correctness proof

Dafny — Features

Foundation: Imperative language with verification built in

Logic: Classical; Z3 SMT solver discharges all goals

Philosophy: code and spec live in the same file — no separate proof script

Key constructs: `requires`, `ensures`, `invariant`, `decreases`, ghost variables

```
method BinarySearch(
    a: array<int>, key: int)
    returns (index: int)
  requires forall i, j ::
    0 <= i < j < a.Length
      ==> a[i] <= a[j]
  ensures index >= 0
      ==> a[index] == key
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 <= lo <= hi
  { var mid := (lo + hi) / 2;
    if      a[mid] < key then lo := mid + 1;
    else if a[mid] > key then hi := mid;
    else return mid; }
  return -1;
}
```

Good for: algorithm proofs, industrial software verification, accessible entry point

Isabelle/HOL — Features

Foundation: Higher-Order Logic (HOL) — classical

Logic: Classical; propositions are *not* types (separate layers)

Automation: **Sledgehammer** calls external ATPs (E, Vampire, Z3, CVC4) and closes most goals in seconds

Archive of Formal Proofs: 700+ vetted entries

```
theorem rev_rev [simp]:
  "rev (rev xs) = xs"
proof (induct xs)
  case Nil
  show ?case by simp
next
  case (Cons a list)
  then show ?case by simp
qed

-- Sledgehammer closes goals:
theorem add_comm:
  "(a :: nat) + b = b + a"
  by sledgehammer
```

Good for: OS and microkernel verification, classical mathematics, strongest automation for first-order reasoning

Isabelle/HOL — seL4

seL4 (Klein et al., 2009) — the most important verified software system ever deployed.

What was verified:

- A complete OS **microkernel** (8,700 lines of C)
- Functional correctness: C code matches the abstract specification
- Memory safety: no buffer overflows, no use-after-free
- Information flow: no data leaks across security domains

Where it runs today:

- DARPA autonomous vehicles
- Boeing military drones
- Nuclear-sector safety systems
- Aviation communication hardware

The classical logic trade-off: Isabelle's HOL means $\neg\neg P = P$ everywhere. Unlike Lean 4, you cannot extract constructive computational content from Isabelle proofs in general.

Verification Landscape: At a Glance

Tool	Logic	Automation	Sweet spot	Notable work
Coq / Rocq	CIC, intuit.	Tactics + Ltac	Verified compilers	CompCert, Iris
Agda	MLTT, intuit.	Pattern matching	PL theory, HoTT	Cubical
F*	Dep. types + effects	SMT (Z3)	Verified crypto	HACL*
Dafny	Classical HOL	SMT (Z3) heavy	Algorithm proofs	Verified algorithms
Isabelle	HOL, classical	Sledgehammer	OS and hardware	seL4
Lean 4	**CIC + DTT**	**Tactics + simp**	**Math + PL**	**Mathlib**

Observation: every tool formalises the same intuition — types are propositions, terms are proofs. Rust bakes this idea into a systems language at scale.

Section 3

A Rust Primer for Lean Programmers

Why Rust?

From without.boats (paraphrasing Graydon Hoare):

> *"Type systems are the parts of formal methods that we've figured out how to make easy. That is, a type system is a formal (and hopefully sound) static analysis technique which is automatically applied to your program to verify aspects of its behaviour."*

Rust shipped a type system that *statically* prevents entire classes of safety bugs — not as a research prototype, but at Mozilla, Linux, Android, AWS, and the Windows kernel.

- Lean 4's type checker *proves theorems*
- Rust's borrow checker *proves memory-safety properties*
- Both are implementations of type theory
- Aeneas: Rust $\longrightarrow$ Lean4 directly

Variables and Mutability

Variables are **immutable by default**. Mutation is opt-in.

```
let x = 5;
// x = 6;          // ERROR: cannot assign twice to immutable variable

let mut y = 5;     // opt in to mutation explicitly
y = 6;             // OK

// Shadowing: a new binding, not a mutation
let x = x + 1;    // new x shadows old x
let x = x * 2;
println!("{}", x); // 12
```

A function taking `&x` cannot change `x` unless it takes `&mut x` — and the type signature tells you which. There is nowhere to hide.

In Lean, everything is immutable by definition. Rust gives you the same guarantee *by default*, with controlled opt-in.

Structs

Rust

```rust
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    fn distance(&self,
                other: &Point) -> f64 {
        let dx = self.x - other.x;
        let dy = self.y - other.y;
        (dx*dx + dy*dy).sqrt()
    } // &self = shared borrow

    fn translate(&mut self,
                 dx: f64, dy: f64) {
        self.x += dx;
        self.y += dy;
    } // &mut self = mutating method
}
```

Lean 4 analogue

```lean
structure Point where
  x : Float
  y : Float
def Point.origin : Point :=
  { x := 0.0, y := 0.0 }
def Point.distance
    (p q : Point) : Float :=
  let dx := p.x - q.x
  let dy := p.y - q.y
  Float.sqrt (dx*dx + dy*dy)

-- No mutation in Lean: return a new Point
↪ instead
def Point.translate
    (p : Point)
    (dx dy : Float) : Point :=
  { x := p.x + dx,
    y := p.y + dy }
```

The `self` receiver encodes *ownership* explicitly in every method signature.

Enums

Rust

```
enum Shape {
    Circle { radius: f64 },
    Rect   { w: f64, h: f64 },
    Triangle(f64, f64, f64),
}
```

Each variant carries different data. This is a **sum type** (tagged union).

Lean 4 analogue

```
inductive Shape where
  | circle (radius : Float)
  | rect   (w h : Float)
  | tri    (a b c : Float)
```

Structurally identical.
Rust `enum` = Lean `inductive`.

Both reject non-exhaustive pattern matches at compile time.

Pattern Matching on Enums

Rust

```rust
fn area(s: &Shape) -> f64 {
    match s {
        Shape::Circle { radius: r } =>
            std::f64::consts::PI * r * r,

        Shape::Rect { w, h } =>
            w * h,

        Shape::Triangle(a, b, c) => {
            let sp = (a + b + c) / 2.0;
            (sp*(sp-a)*(sp-b)*(sp-c))
                .sqrt()
        }
    }
}
```

Non-exhaustive matches are rejected. No case can be forgotten.

Lean 4 analogue

```
def area : Shape -> Float
  | .circle r =>
      Float.pi * r * r

  | .rect w h =>
      w * h

  | .tri a b c =>
      let sp := (a + b + c) / 2
      Float.sqrt
        (sp*(sp-a)*(sp-b)*(sp-c))
```

Identical exhaustiveness guarantee.

`Option` — No Null Pointers

Rust

```rust
// Built into the language
enum Option<T> { Some(T), None }

fn safe_div(a: f64, b: f64)
    -> Option<f64> {
    if b == 0.0 { None }
    else { Some(a / b) }
}

fn main() {
    match safe_div(10.0, 2.0) {
        Some(v) => println!("{}", v),
        None    => println!("div/0"),
    }
    // Forgetting None is a
    // COMPILE ERROR
}
```

Rust has **no null pointers**. Absence (or not) is encoded in the type.

Lean 4 analogue

```
inductive Option (alpha : Type) where
  | some : alpha -> Option alpha
  | none : Option alpha

def safeDiv (a b : Float)
    : Option Float :=
  if b == 0.0 then none
  else some (a / b)

-- Pattern match is exhaustive.
-- Forgetting none is also
-- a compile error in Lean.
```

Fallibility is encoded in the type, not hidden in a runtime exception. Curry-Howard made practical.

`Result` — No Exceptions

Rust

```rust
enum Result<T, E> { Ok(T), Err(E) }

// The ? operator propagates errors
// automatically (like »= for Except)
fn parse_and_double(s: &str)
    -> Result<i32, String>
{
    let n: i32 = s.parse()
        .map_err(|e| e.to_string())?;
    Ok(n * 2)
}

match parse_and_double("21") {
    Ok(v)  => println!("{}", v), // 42
    Err(e) => println!("{e}"),
}
```

Rust has **no exceptions**. Errors are values.

Lean 4 analogue

```
inductive Except
    (eps alpha : Type) where
  | ok    : alpha ->
              Except eps alpha
  | error : eps ->
              Except eps alpha

def parseAndDouble (s : String)
    : Except String Int :=
  match s.toInt? with
  | none   =>
      .error "not a number"
  | some n => .ok (n * 2)

-- do-notation threads errors
def example : Except String Int
    := do
  let n <- parseAndDouble "21"
  return n + 1
```

Traits

A **trait** is a named collection of method signatures. Implementing a trait is a promise that a type satisfies a contract.

```
// Define a trait
trait Area {
    fn area(&self)      -> f64;
    fn perimeter(&self) -> f64;
}

// Implement for a type
impl Area for Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Circle { radius: r } => std::f64::consts::PI * r * r,
            Shape::Rect { w, h }        => w * h,
            Shape::Triangle(a, b, c)    => { /* whatever */ 0.0 }
        }
    }
    fn perimeter(&self) -> f64 { /* ...stuff... */ 0.0 }
}
```

Trait Bounds on Generic Functions

Rust

```rust
// T must implement Area
fn print_area<T: Area>(s: &T) {
    println!("{:.2}", s.area());
}

// Multiple bounds with where clause
fn compare_shapes<T, U>(
    a: &T,
    b: &U,
) -> std::cmp::Ordering
where
    T: Area,
    U: Area,
{
    a.area().total_cmp(&b.area())
}
```

Lean 4 analogue

```lean
class HasArea (alpha : Type) where
  area : alpha -> Float

instance : HasArea Shape where
  area := Shape.area

def printArea [HasArea alpha]
    (s : alpha) : IO Unit :=
  println! "{:.2}"
    (HasArea.area s)

def compareShapes
    [HasArea alpha] [HasArea beta]
    (a : alpha) (b : beta)
    : Ordering :=
  compare (HasArea.area a)
          (HasArea.area b)
```

Trait bounds constrain what types a generic function accepts.

Ownership and Move Semantics

In Rust, **using a value = consuming it** by default.

```
let s1 = String::from("hello");  // s1 owns the heap data
let s2 = s1;                     // ownership MOVES to s2

// println!("{}", s1);           // ERROR: use of moved value
println!("{}", s2);              // OK

// Functions also consume their arguments
fn take(s: String) {
    println!("{}", s);
} // s is dropped here --- heap memory freed automatically

let greeting = String::from("hello");
take(greeting);
// println!("{}", greeting);     // ERROR: greeting was moved into take()

// To keep both alive, .clone() makes an explicit deep copy
let s3 = s2.clone();
println!("{} {}", s2, s3);       // both OK
```

Section 4

The Problem: Mutable Aliased State

References Are Like Jumps

C.A.R. Hoare, 1974:

> *"References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover."*

If two names can refer to the same memory location (*aliasing*), and that memory can be mutated, then *local reasoning breaks down*:

- You update `x` and `y` unexpectedly changes — *spooky action at a distance*
- Iterator invalidation, use-after-free, data races
- Double-free vulnerabilities, UAF CVEs

Mutable aliased state is not just a performance concern. It is a *logical* problem.

Failed Attempts: OOP and Pure FP

Object-Orientation (1970s)

- Encapsulate mutable state inside objects
- Methods enforce invariants

Why it falls short:
An encapsulated aliased mutable reference is *still* an aliased mutable reference. A method on one alias can violate an invariant that another alias depends on.

Joe Armstrong: "You wanted a banana but what you got was the gorilla holding it and the entire jungle."

Pure Functional Programming (1980s–90s)

- Eliminate mutation entirely
- Alias freely; nothing can change

What works:
Without mutation, aliasing is harmless. Concurrent programs can share state safely.

What falls short:
Real programs do I/O and interact with stateful hardware. Very hard to match C performance on inherently imperative hardware.

Failed Attempts: Monads

Monads (Haskell, 1990s)

- Model effects compositionally in types
- Separate pure and effectful code
- State monad sequences mutations explicitly

What monads achieve:

- A sound solution to the I/O problem
- Pure functions are easy to verify

What monads don't achieve:

- Still requires manual aliasing reasoning
- Famously hard to teach
- Overhead on imperative hardware

From without.boats:

> *"Monads are a clever way to show you can program without mutation; lifetimes are an even cleverer way to show you can just use mutation."*

Rust's approach:

- Keep mutation (it is useful and efficient)
- Statically forbid *mutable aliasing*
- References are mutable or aliased, never both
- The type checker enforces this as a *theorem* about every program

Section 5

Substructural Types

Structural Rules in Logic

You already know these from Lean's sequent calculus. **Structural rules** govern how the hypothesis context Γ may be manipulated — they are meta-rules *about* reasoning, not about specific propositions.

Weakening:

$$\frac{\Gamma \vdash A}{\Gamma, B \vdash A}$$

"You can *ignore* hypotheses."

In a type system: a variable may go *unused*. You can drop a value.

Contraction:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

"You can *duplicate* a hypothesis."

In a type system: a variable may be *used more than once*. You can copy a value.

In Lean 4: **both rules hold**. Hypotheses can be ignored and reused freely.

Rust *restricts contraction*. This is the key.

The Substructural Zoo

Selectively removing structural rules yields different *resource disciplines* — type systems that track how many times a value is used:

System	Weakening?	Contraction?	How many times?
Normal types	✓	✓	Any number
Affine types	✓	×	**At most once** (can drop, cannot copy)
Relevant types	×	✓	**At least once** (must use, can copy)
Linear types	×	×	**Exactly once** (must use, cannot copy)

These correspond to affine logic, relevant logic, and linear logic respectively. Each has a Curry-Howard interpretation in terms of resource consumption.

Where Does Rust Sit?

Most Rust types are affine

Weakening ✓: you can drop a value (let it go out of scope without using it).
Contraction ×: you cannot duplicate a value (assign the same `String` to two variables).

Ownership is the programmer-facing metaphor for *affine types*. Moving a value = consuming a linear hypothesis.

The `Copy` exception

Types implementing `Copy` (`i32`, `bool`, `f64`, `char`…) are normal types: contraction is allowed via a bitwise copy.

`String` is *not* `Copy`: it owns a heap allocation. Duplicating it requires a new allocation, made explicit with `.clone()`.

`Clone` = contraction, made *opt-in* so the programmer is aware of the cost.

Rust currently lacks true *linear* types (types that *must* be used). This is an active area of language design — the proposal is called "must-move types" or "undroppable types".

Ownership and RAII (see Note for Rust Nerds 1, slide 72)

Because the type system guarantees **exactly one owner**, Rust calls the destructor (`Drop`) automatically when that owner goes out of scope. This is **RAII**.

```
fn read_file() {
    let f = File::open("data.txt").unwrap();
    process(&f);
} // f goes out of scope here
  // close() is called AUTOMATICALLY
  // No try/finally, no risk of forgetting

fn with_lock(m: &Mutex<i32>) {
    let guard = m.lock().unwrap();
    *guard += 1;
} // guard goes out of scope here
  // mutex is RELEASED automatically
  // No deadlock from a forgotten unlock()
```

Ownership types are *session types* with a default transition. The destructor is the "default" state transition called when a value is not otherwise consumed. The type system enforces a liveness property: the file *will* be closed; the lock *will* be released.

Section 6

Borrowing and Lifetimes

The Central Invariant

The rule the borrow checker enforces:

At any point in the program, for any value v:

$$\underbrace{\text{any number of shared references}}_{\text{read-only}} \quad \mathbf{XOR} \quad \underbrace{\text{exactly one mutable reference}}_{\text{read-write}}$$

If you hold a mutable reference, *no other reference to the same value exists*. Therefore mutation cannot affect any other alias. Local reasoning is restored: you know exactly which state can change.

Spooky action at a distance becomes impossible *by construction*.

Shared References `&T`

Properties

- Any number may coexist simultaneously
- Read-only access to the data
- Type written `&'a T`

Type-theory reading

A shared reference is a *normal* type: it has contraction (can be copied to create another shared reference) and weakening (can be dropped).
This corresponds to the intuitionistic persistent modality $\Box A$ in linear logic — "always available, never consumed".

Intuition

A `&String` is like a library book on a shared shelf. Many people can read it at once. No one can write in it. No one "owns" it.

Common uses

- Passing data without giving it away
- Inspecting state without locking it
- Sharing read-only configuration across threads

Mutable References `&mut T`

Properties

- *Exactly one* may exist at a time
- Full read/write access
- While it exists, *no other reference* (shared or mutable) to the same value may exist
- Type written `&'a mut T`

Type-theory reading

A mutable reference is an *affine* type: it has weakening (can be dropped), but not contraction (cannot be aliased — that would violate exclusivity). This corresponds to a linear resource in linear logic.

Intuition

A `&mut String` is like checking a book out of the library. Only you have it. No one else can read it while you hold it. You must return it before anyone else can access it.

Common uses

- Modifying data in place
- Filling a buffer
- Sorting a collection
- Initialising a struct's fields

Borrowing: The Rules in Practice

```rust
fn print_len(s: &String) { println!("{}", s.len()); }

fn make_louder(s: &mut String) { s.push_str("!!!"); }

fn main() {
    let mut text = String::from("Hello");

    print_len(&text);              // shared borrow --- OK
    print_len(&text);              // another shared borrow --- OK
    make_louder(&mut text);        // mutable borrow --- OK (no shared borrows active)
    print_len(&text);              // shared borrow again --- "Hello!!!"

    let r1 = &text;
    let r2 = &text;                // two shared borrows: fine
    // make_louder(&mut text);     // COMPILE ERROR: mutable borrow while
    //                             // r1, r2 are live
    println!("{} {}", r1, r2);     // r1, r2 end here
    make_louder(&mut text);        // NOW OK: r1 and r2 are gone
}
```

What Is a Lifetime?

A common misconception: lifetimes are about how long something "lives" in wall-clock time.

What they actually are: a lifetime `'a` is a *region variable* — a name for a *scope* during which a reference is valid. Lifetimes express **scope-containment constraints**.

Formally (Oxide, Weiss et al. 2019):

```
&'a T
```

is a reference valid within region `'a`.

`'b: 'a` means `'b` *outlives* `'a`. The borrow checker solves a constraint system over this partial order.

Intuitively:
A lifetime is a *proof* that a reference is still valid. When the referenced scope ends, the proof expires.

The borrow checker is a *decision procedure* for region-containment constraints. It rejects programs for which it cannot prove that all references are used within valid regions.

Lifetimes in Function Signatures

Rust — explicit lifetime

```rust
// 'a ties the output's validity
// to the minimum of both inputs.
fn longest<'a>(
    x: &'a str,
    y: &'a str,
) -> &'a str {
    if x.len() > y.len() { x }
    else { y }
}

// Without 'a: COMPILE ERROR
// "missing lifetime specifier"
// The compiler cannot determine
// which input the output borrows.
```

When a function returns a reference, the compiler must know *which input* it borrows from.

Lean 4 — dependent type instead

```
-- No aliasing in Lean:
-- no lifetime needed.

def longest (x y : String)
    : String :=
  if x.length >= y.length
  then x else y

-- To prove the output is
-- one of the inputs:
def longestSpec (x y : String)
    : { s : String //
        s = x ∨ s = y } :=
  if x.length >= y.length
  then ⟨x, Or.inl rfl⟩
  else ⟨y, Or.inr rfl⟩
```

In Lean the invariant is a proposition. In Rust it is a lifetime constraint. Both are checked statically.

Iterator Invalidation: C++ vs. Rust

C++ — undefined behaviour

```cpp
std::vector<int> v = {1, 2, 3};

for (auto& x : v) {
    if (x == 2)
        v.push_back(42);
    // UNDEFINED BEHAVIOUR:
    // push_back may reallocate,
    // invalidating the iterator.
    // This has been a real
    // CVE class for decades.
}
```

One of the most common C++ bug classes.

Rust — ”complains” at compile-time

```rust
let mut v = vec![1, 2, 3];

for x in &v {       // shared borrow
    if *x == 2 {
        v.push(42);
        // COMPILE ERROR:
        // cannot borrow `v` as
        // mutable because it is
        // also borrowed as immutable
        // (by the for loop)
    }
}
```

Rust prevents this thing at compile-time!

The `for` loop holds `&v`; `push` needs `&mut v`. The aliasing XOR mutation rule forbids both simultaneously. The bug *cannot occur by construction*.

The Key Theorem: RustBelt

How do we know the borrow checker is correct?

RustBelt (Jung et al., POPL 2018) proved in Rocq that Rust's type system is *semantically sound*.

What was proved:

1. Safe Rust is memory-safe: no use-after-free, no double-free, no data races
2. Key standard-library types are sound *despite* using `unsafe` internally: Mutex, Arc, Cell, Vec
3. Lifetime logic correctly models the borrow checker

Tool: Iris (a separation logic framework in Coq), extended with a *lifetime logic* where borrows are propositions with temporal structure.

The formal condition:
The typing context in λ_{Rust} is *substructural*:

$$p \trianglelefteq \tau$$

may be duplicated only if τ `copy` holds — i.e. only if `T: Copy`.

This is the contraction rule from our table, now embedded in a Rocq proof about Rust's operational semantics.

Section 7

Traits as Proof Obligations

Traits Are Propositions on Types

You already know that in Lean, a type class instance is a *proof* that a type satisfies a predicate. The same is true in Rust.

In Lean 4

`[Inhabited alpha]` in a function signature is a proof that `alpha` has a default element.

The instance `instance : Inhabited Nat` is a proof term. The type class system is a proof-search procedure.

In Rust

`T: Area` in a generic bound is evidence that `T` has an `area` method. The compiler checks this evidence at every call site.

`impl Area for Circle` is the proof term. Trait resolution is Rust's proof-search procedure.

Syntactically different; logically identical. Both are instances of propositions-as-types applied to predicates over types.

Safety Marker Traits: Send and Sync

Some traits in Rust are *safety proofs* that the compiler checks mechanically:

T: Send

"It is safe to transfer ownership of T across a thread boundary."

Formally (RustBelt): T: Send ≈ the ownership predicate of T does not depend on the current thread identifier.

T: Sync

"It is safe for two threads to hold &T simultaneously."

Formally: T: Sync ≈ &T: Send.

```rust
fn send_to_thread<T: Send>(x: T) {
    std::thread::spawn(move || {
        println!("{:?}", x);
    });
}

// Raw pointers are NOT Send.
// The compiler rejects this:
use std::ptr;
let p = ptr::null::<i32>();
// std::thread::spawn(move || {
//   println!("{:?}", p);
// });
// ERROR: *const i32 is not Send
// (raw pointers can alias)
```

Phantom Types: Type-Level State Machines (see Note for Rust Nerds 2, slide 73)

Types can carry *compile-time proof information* at zero runtime cost using phantom type parameters (e.g. type encodes a protocol, compiler = protocol enforcer):

```rust
use std::marker::PhantomData;

struct Locked;     // type-level tags --- zero runtime bytes
struct Unlocked;

struct TypedMutex<T, State> {
    data:   T,
    _state: PhantomData<State>,  // zero-cost compile-time marker
}

impl<T> TypedMutex<T, Locked> {
    fn unlock(self) -> TypedMutex<T, Unlocked> { /* ...stuff... */ }
}

impl<T> TypedMutex<T, Unlocked> {
    fn lock(self) -> TypedMutex<T, Locked>   { /* ...stuff... */ }
    fn get(&self) -> &T { &self.data }  // only callable when Unlocked
}
// Calling get() on TypedMutex<T, Locked> is a COMPILE-TIME TYPE ERROR
```

Rust vs. Lean: Expressiveness

Rust: phantom tag approach

```rust
// Encode "non-empty list" in the type
struct NonEmpty<T>(Vec<T>);

impl<T> NonEmpty<T> {
    fn new(x: T, rest: Vec<T>)
        -> Self {
        let mut v = vec![x];
        v.extend(rest);
        NonEmpty(v)
    }
    // No runtime check needed.
    fn head(&self) -> &T {
        &self.0[0]
    }
}
```

Good, but Rust can't say "exactly n elements" for a *runtime* n.

Lean 4: dependent types

```
-- The LENGTH lives in the type.
def Vec' (alpha : Type) (n : Nat) :=
  { l : List alpha //
    l.length = n }
-- head is TOTAL: the nil case
-- is logically impossible.
def head' {alpha : Type} {n : Nat}
    (v : Vec' alpha (n+1))
    : alpha :=
  match v.val with
  | []     =>
      absurd v.prop (by simp)
  | x :: _ => x
```

Values appear inside types. `n` is a runtime value that is also a type-level constraint.

Rust's type system is less expressive than Lean's (no full dependent types), but the borrow checker adds *resource reasoning* that Lean does not have natively.

Section 8

Rust in Practice

Iterators as Lazy Pipelines

Rust iterators are **lazy**: they describe a transformation without executing it. Execution only happens when a *consuming adapter* is called.

```rust
// Nothing runs yet --- just a description of the pipeline
let pipeline = (1..=1000)
    .filter(|x| x % 2 == 0)    // keep even numbers
    .map(|x| x * x)            // square each one
    .take(5);                  // stop after five

// Execute by consuming
let result: Vec<i32> = pipeline.collect();
// [4, 16, 36, 64, 100]

// Other common consumers
let sum: i32  = (1..=100).sum();                          // 5050
let count     = (1..=100).filter(|x| x % 3 == 0).count(); // 33
let first     = (1..).find(|x| x * x > 50);               // Some(8)
```

Notice the pattern: laziness means no intermediate allocations — the chain processes one element at a time, from source to consumer.

Iterator Adapters

The iterator API covers the same ground as Lean's `List` library, but in a lazy, composable style.

```rust
let numbers: Vec<i32> = (1..=10).collect();

// Sum of squares of even numbers
let result: i32 = numbers.iter()
    .filter(|&&x| x % 2 == 0)
    .map(|&x| x * x)
    .sum();
// 4 + 16 + 36 + 64 + 100 = 220

// Chain across different adapter types
let odds_as_strings: Vec<String> = numbers.iter()
    .filter(|&&x| x % 2 != 0)
    .map(|x| x.to_string())
    .collect();
// ["1", "3", "5", "7", "9"]

// Flatten nested structure
let nested = vec![vec![1, 2], vec![3, 4], vec![5]];
let flat: Vec<i32> = nested.into_iter().flatten().collect();
// [1, 2, 3, 4, 5]
```

Closures and Environment Capture

Closures capture variables from their enclosing scope.
The move keyword transfers ownership into the closure.

```
// Capture by reference (default): closure borrows `threshold`
let threshold = 5;
let above: Vec<i32> = (1..=10)
    .filter(|&x| x > threshold)
    .collect(); // Value: [6, 7, 8, 9, 10]

// `move` closures take ownership --- useful for returning closures
//  or sending them to another thread
fn make_adder(n: i32) -> impl Fn(i32) -> i32 {
    move |x| x + n   // n is moved into the closure
}

let add5  = make_adder(5);
let add10 = make_adder(10);
println!("{}", add5(3));    // 8
println!("{}", add10(3));   // 13
println!("{}", add5(7));    // 12  (add5 is reusable --- Fn, not FnOnce)
```

Notice the pattern: `make_adder` is just partial application, expressed through closure capture rather than currying.

Error Propagation with ?

The ? operator threads `Result` through a function, eliminating boilerplate match chains.

```rust
use std::fs;
use std::num::ParseIntError;

// Two possible failure modes, unified under one error type
fn read_and_double(path: &str) -> Result<i32, Box<dyn std::error::Error» {
    let content = fs::read_to_string(path)?;  // IO error  -> return Err early
    let n: i32  = content.trim().parse()?;    // parse error -> return Err early
    Ok(n * 2)
}

// What `?` desugars to:
//   match expr {
//       Ok(x)  => x,                     // continue with x
//       Err(e) => return Err(e.into()),  // propagate up
//   }

// With `?` the happy path reads linearly, top to bottom.
// Compare: Lean's do-notation for Except/Option.
```

The Newtype Pattern

Wrapping a primitive type in a struct creates a **distinct type at zero runtime cost**. The compiler refuses to mix them up.

```
struct Metres(f64);
struct Kilograms(f64);
struct Seconds(f64);

fn speed(distance: Metres, time: Seconds) -> f64 {
    distance.0 / time.0
}

// Compile error --- units cannot be confused:
// speed(Kilograms(70.0), Seconds(10.0));
// ERROR: expected `Metres`, found `Kilograms`

// Same memory layout as f64, so no overhead.
// But the type system catches the mistake Mars Climate
// Orbiter made (mixed metric/imperial, $327M loss, 1999).
```

This is the same idea as Lean's `structure` wrappers — using types to eliminate a whole class of bugs statically, with no runtime cost.

The Builder Pattern

Rust's move semantics make the builder pattern natural and safe: each setter consumes the builder and returns a new one, preventing partially-constructed objects from escaping.

```
struct Request { url: String, method: String, timeout_ms: u64 }

struct RequestBuilder { url: String, method: String, timeout_ms: u64 }

impl RequestBuilder {
    fn new(url: &str) -> Self {
        Self { url: url.into(), method: "GET".into(), timeout_ms: 5000 }
    }
    fn method(mut self, m: &str)  -> Self { self.method     = m.into(); self }
    fn timeout(mut self, ms: u64) -> Self { self.timeout_ms = ms;       self }
    fn build(self) -> Request {
        Request { url: self.url, method: self.method, timeout_ms: self.timeout_ms }
    }
}

let req = RequestBuilder::new("https://api.example.com/data")
    .method("POST")
    .timeout(3000)
    .build();
```

Section 9

Rust and the Verification Ecosystem

The Framing Problem

A central challenge in program verification:

The **frame** of an operation is the part of memory it does *not* touch. If you can identify the frame, you know which existing facts remain true after the operation.

Without a type discipline, framing is hard:

```
// Does this code change y?
// In C/Java: IMPOSSIBLE TO KNOW statically
// x and y might point to the same location
fn increment(x: *mut i32) { unsafe { *x += 1; } }
```

Separation logic (O'Hearn, Reynolds, 2001) solves this with the separating conjunction $P * Q$: "P holds on one disjoint heap region, Q on another."

$$\{P * F\}\ \texttt{cmd}\ \{Q * F\} \quad \text{if cmd doesn't touch the frame } F$$

Manual separation logic is laborious. Rust automates it.

Rust Automates Separation Logic

From Viper / Prusti (2019):

Rust's borrow checker already enforces separation — a mutable reference is exclusive by definition. The type system is doing the separation logic for you.

Traditional separation logic

- Annotate every operation with ownership predicates
- Manually discharge frame conditions
- Prove no aliasing at every step
- Very tedious; doesn't scale easily

Rust + verification tool

- Separation for free (borrow checker)
- Tools only need to check *functional correctness*
- Memory reasoning is already done
- This is why Rust is so attractive as a verification target

We could rely on the Rust type system to guarantee memory safety and separation for us, freeing us from the need to do that in separation logic

The Rust Verification Toolchain

Aeneas (Ho, Protzenko, Fromherz, ICFP 2022)

- Rust $\rightarrow$ **Lean 4** (also F*, Coq)
- Borrows become pure function arguments and return values
- Memory reasoning eliminated for safe Rust
- Prove properties using Lean 4 tactics

RefinedRust (Gäher et al., PLDI 2024)

- Handles full complexity of mutable borrows
- Built on Iris / RustBelt in Coq
- Targets production-grade code

Prusti (ETH Zürich)

- Annotation-based: `#[requires(...stuff...)]`
- Discharges via Viper / Z3
- Closest to Dafny for Rust

Flux (Lehmann et al.)

- Refinement types inside Rust
- `i32{v: v > 0}` in signatures
- Strong SMT automation

Verus (Microsoft Research)

- Ghost variables and spec functions
- SMT-based; close to Dafny in style

Aeneas: The Key Idea

The challenge: mutable borrows create a non-trivial memory model — how do we translate Rust to a pure language?

Aeneas's insight: a mutable borrow is a *temporary transfer of ownership*. When the borrow ends, ownership returns to the original binding — possibly with a new value.

Model this as a function that takes the current value and returns both its result and the new value:

Rust	⟶	Lean 4
`fn f(x: &mut T) -> R`	⟶	`def f (x : T) : Result (R × T)`

No memory, no pointers, no aliasing. The entire borrow system collapses into pure function composition. Lean tactics can then prove properties of these functions.

Aeneas: A Worked Example

Rust with mutable borrow

```rust
fn increment(x: &mut i32) {
    *x += 1;
}

fn double_increment(x: &mut i32) {
    increment(x);
    increment(x);
}

fn main() {
    let mut v = 5;
    double_increment(&mut v);
    println!("{}", v); // 7
}
```

The borrow `&mut v` passes temporary exclusive ownership through two function calls.

Aeneas translation (Lean 4)

```
-- Borrow becomes (result, new value)
def increment (x : Int)
    : Result (Unit × Int) :=
  ok ((), x + 1)

def doubleIncrement (x : Int)
    : Result (Unit × Int) := do
  let (_, x1) <- increment x
  let (_, x2) <- increment x1
  ok ((), x2)

-- Now prove a property:
theorem doubleIncrement_spec
    (x : Int)
    : doubleIncrement x
      = ok ((), x + 2) := by
  simp [doubleIncrement,
        increment]
```

Section 10

The Curry–Howard Connection

Rust Through the Curry–Howard Lens

Types $\approx$ propositions, terms $\approx$ proofs. Rust adds a *resource-sensitive* dimension.

Rust	Lean 4	Logic
`fn(T) -> U`	`T -> U`	$T \Rightarrow U$
`(T, U)`	`T × U`	$T \wedge U$
`enum Either{L(T), R(U)}`	`T ⊕ U`	$T \vee U$
`!` (never type)	`Empty`	$\bot$
`()`	`Unit`	$\top$
Move semantics	Linear hypothesis	Linear A
`&'a T`	Persistent resource in a	$\square A$
`&'a mut T`	Exclusive resource in a	Linear A
Lifetime `'a: 'b`	$a \supseteq b$	Region ordering
`T: Send`	Proof of thread-safety	Predicate instance
`T: Copy`	Contraction allowed	Normal type

What Lean 4 Has

Strengths

- Full dependent types
- Propositions are types, all the way down
- Proof terms are programs
- The kernel checks everything
- Pure: no aliasing, no mutation by default

What Lean lacks natively

- Resource tracking: no notion of “this value has been used”
- Region analysis: no lifetimes
- Memory model: no built-in heap reasoning

These can be modelled (e.g. with separation logic in Lean), but they are not part of the type checker itself.

What Rust Has

Strengths

- Affine types (ownership, move semantics)
- Region variables (lifetimes)
- Borrow checker: automated decision procedure for aliasing constraints
- Safe mutation without aliasing
- Memory management without GC

What Rust lacks

- Full dependent types
- Proof terms / propositions as types
- Cannot express “sorted list” or “n-element vector” for a runtime n

They are complementary. Aeneas, Prusti, and Verus bridge the two worlds, letting you write in Rust and verify in Lean or Coq.

Section 11

Summary and Looking Ahead

What We've Established: The Theory

- Ownership = affine types (weakening yes, contraction no)
- `Copy` types = normal types (both structural rules)
- Borrowing = temporary exclusive or shared access, tracked by region variables
- Lifetimes = region variables with a partial order; the borrow checker solves the constraint system
- Traits = type class instances = proofs that a type satisfies a predicate
- Rust's type system = a fragment of intuitionistic linear logic, made practical
- The correspondence to Lean is exact at every level: `enum`/`inductive`, trait bounds/class instances, `match`/pattern match

What We've Established: The Practice

- Every Rust program that compiles is a proof of memory safety and data-race freedom
- This is formally proved in Coq (RustBelt, POPL 2018) using the Iris separation logic framework
- Aeneas translates Rust programs to Lean 4 for functional correctness proofs, eliminating memory reasoning entirely
- Tools like Prusti and Verus add SMT-based specification directly in Rust, close to Dafny in style
- The borrow checker automates the frame condition in separation logic — a task that is manual and laborious in every other verification approach

The One-Line Summary

"Monads are a clever way to show you can program without mutation. Lifetimes are an even cleverer way to show you can just use mutation."

— without.boats

Rust didn't add lifetimes to avoid garbage collection. It added rules about shared mutable state — and avoiding GC was a consequence. The rules are a good idea regardless.

Next Week: Project Showcase

Week 10 — Course wrap-up and student projects

Readings before then:

Core

- Son Ho et al., *Aeneas: Rust Verification by Functional Translation*, ICFP 2022
- `lean-lang.org/use-cases/aeneas`
- without.boats, “Ownership” and “References Are Like Jumps” (both linked on the course website)

Questions & Discussion

Questions?

Join our community:

Discord: Link on website

WhatsApp: Link on website

Website: `https://danieldia-dev.github.io/proofs/`

Email: `dmd13@mail.aub.edu`

Rust Nerd Note #1: Destructors Are Not Guaranteed

On the RAII slide: the framing "`Drop` is called *automatically*" is true in the common case, but not a language guarantee.

```
let f = File::open("data.txt").unwrap();
std::mem::forget(f);   // safe, compiles fine ---
                       // file handle leaks forever
```

`std::mem::forget` is **not** `unsafe`. From the docs: *"Any resources the value manages [...] will linger forever in an unreachable state [...] `forget` is not marked as `unsafe` because Rust's safety guarantees do not include a guarantee that destructors will always run."*

Other leaking mechanisms

- Rc cycles (reference-count never reaches zero)
- Box::leak (intentional static lifetime)
- Process abort (std::process::abort)

What *is* guaranteed: No use-after-free, no double-free, no data races.
Resource *cleanup* is a liveness property; memory *safety* is a safety property. Rust guarantees the latter, not the former.

Rust Nerd Note #2: Phantom Types and Runtime Checks

On the `NonEmpty` slide: `NonEmpty::<T>::head` still performs a runtime bounds check on the underlying `Vec`. The type system prevents calling `head` on an empty value, but `a[0]` inside the body is not aware of that invariant — the compiler will still emit a bounds check. To truly eliminate the check you must enter `unsafe` and uphold the invariant manually:

The stdlib does this properly with `NonZero`

```
// Division by NonZero<u8> compiles
// to a single `div` instruction ---
// no zero-check emitted.
let den = NonZero::<u8>::new(4).unwrap();
let result: u8 = 12u8 / den;
```

The invariant lives in the type; the compiler uses `std::hint::assert_unchecked` internally to elide the check.

The general lesson

```
fn head(v: &NonEmpty<i32>) -> &i32 {
    // NonEmpty guarantees v.0 is non-empty,
    ↪  so index 0 is always valid.
    unsafe { v.0.get_unchecked(0) }
}
```

Types can *prevent* invalid states, but eliminating the *runtime cost* of checking them requires `unsafe` + an explicit proof comment

"Type systems are the parts of formal methods that we've figured out how to make easy."

— Graydon Hoare (paraphrased)

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 9 of 10

Rust as a Proof Assistant

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/