How to Make Mathematicians Into Programmers (And Vice Versa)

Will Crichton

Assistant Professor, Brown University

Mathematicians working on paper

A CLASSIFICATION OF IMMERSIONS OF THE TWO-SPHERE

BY
STEPHEN SMALE

An immersion of one C^1 differentiable manifold in another is a regular map (a C^1 map whose Jacobian is of maximum rank) of the first into the second. A homotopy of an immersion is called regular if at each stage it is regular and if the induced homotopy of the tangent bundle is continuous. Little is known about the general problem of classification of immersions under regular homotopy. Whitney [5] has shown that two immersions of a k-dimensional manifold in an n-dimensional manifold, $n \geqq 2k+2$, are regu-

formalized mathematics

Developers working with weak type systems

```
-        /* Read type and payload length first */
-        hbtype = *p++;
-        n2s(p, payload);
-        pl = p;
-
         if (s->msg_callback)
                 s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                         &s->s3->rrec.data[0], s->s3->rrec.length,
                         s, s->msg_callback_arg);

```

```
         if (s->msg_callback)
                 s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
                         &s->s3->rrec.data[0], s->s3->rrec.length,
                         s, s->msg_callback_arg);

+        /* Read type and payload length first */
+        if (1 + 2 + 16 > s->s3->rrec.length)
+                return 0; /* silently discard */
+        hbtype = *p++;
+        n2s(p, payload);
```

Introduction
1 Loops
2 Local theory of convex integration
3 Global theory of open and ample relations
A Local sphere eversion
B From local to global
Dependency graph

The sphere eversion project

Patrick Massot Oliver Nash Floris van Doorn

formalized mathematics

Project Everest

Papers People In the News Related Projects

Microsoft · Inria · Carnegie Mellon University · Microsoft Research - Inria JOINT CENTRE

We are a team of researchers and engineers from several organizations, including Microsoft Research, Carnegie Mellon University, INRIA, and the MSR-INRIA joint center.

Provably Secure Communication Software

AUTOMATED REASONING

How the Lean language brings math to coding and coding to math

Uses of the functional programming language include formal mathematics, software and hardware verification, AI for math and code synthesis, and math and computer science education.

By Leo de Moura

Share

August 16, 2024

How the Lean language brings math to coding and coding to math

Uses of the functional programming language include formal mathematics, software and hardware verification, AI for math and code synthesis, and math and computer science education.

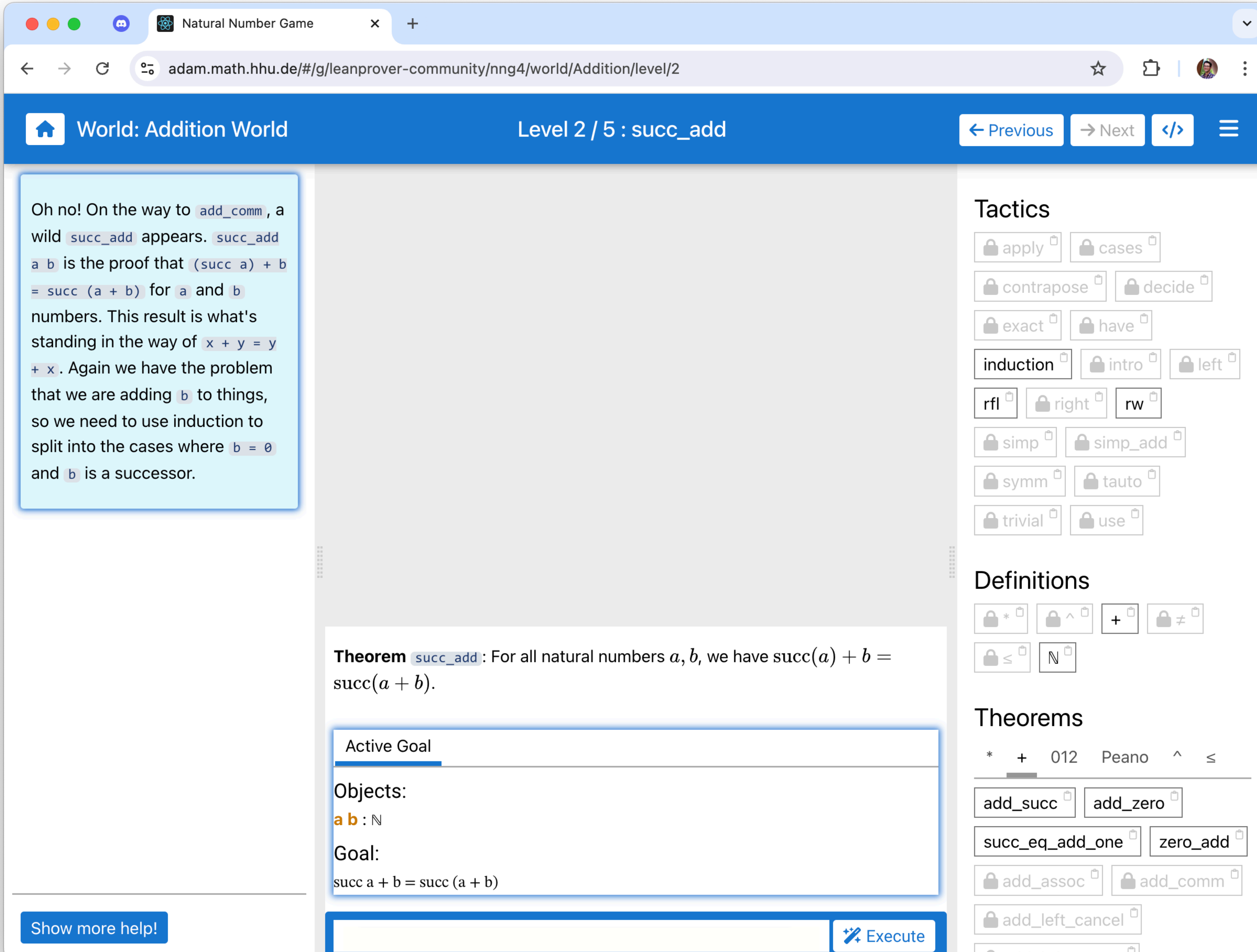
By Leo de Moura

Share

August 16, 2024

How to Make Mathematicians Into Programmers (And Vice Versa)

Learning Lean



https://worrydream.com/LearnableProgramming/

World: Addition World | Level 2 / 5 : succ_add | Previous | Next

Oh no! On the way to `add_comm`, a wild `succ_add` appears. `succ_add a b` is the proof that `(succ a) + b = succ (a + b)` for `a` and `b` numbers. This result is what's standing in the way of `x + y = y + x`. Again we have the problem that we are adding `b` to things, so we need to use induction to split into the cases where `b = 0` and `b` is a successor.

Show more help!

Tasks:

- **What problem am I solving?**
- **What is the state of my solution?**
- **What tactic should I use?**
- **What theorem should I use?**
- **What have I tried already?**

Theorem `succ_add`: For all natural numbers a, b, we have $\mathrm{succ}(a) + b = \mathrm{succ}(a + b)$.

Active Goal

Objects:

a b : ℕ

Goal:

succ a + b = succ (a + b)

Execute

Tactics

apply | cases | contrapose | decide | exact | have | induction | intro | left | rfl | right | rw | simp | simp_add | symm | tauto | trivial | use

Definitions

$*$ | ^ | + | ≠ | ≤ | ℕ

Theorems

$*$ | + | 012 | Peano | ^ | ≤

add_comm | add_succ | add_zero | succ_add | succ_eq_add_one | zero_add | add_assoc | add_left_cancel | add_left_comm | add_left_eq_self

Oh no! On the way to `add_comm`, a wild `succ_add` appears. `succ_add a b` is the proof that `(succ a) + b = succ (a + b)` for `a` and `b` numbers. This result is what's standing in the way of `x + y = y + x`. Again we have the problem that we are adding `b` to things, so we need to use induction to split into the cases where `b = 0` and `b` is a successor.

You might want to think about whether induction on `a` or `b` is the best idea.

What theorem should I use?

Which theorem defines equality over an expression that unifies with a sub-expression of the goal?

Theorem `succ_add`: For all natural numbers a, b, we have $\mathrm{succ}(a) + b = \mathrm{succ}(a + b)$.

Active Goal

Objects:

a b : ℕ

Goal:

$\mathrm{succ}\ a + b = \mathrm{succ}\ (a + b)$

```
induction b with b hb
```

Retry

Active Goal | Goal 2

Objects:

a : ℕ

Goal:

$\mathrm{succ}\ a + 0 = \mathrm{succ}\ (a + 0)$

Execute

Tactics

apply cases contrapose decide exact have induction intro left rfl right rw simp simp_add symm tauto trivial use

Definitions

* ^ + ≠ ≤ ℕ

Theorems

* + 012 Peano ^ ≤

add_comm add_succ add_zero succ_add succ_eq_add_one zero_add add_assoc add_left_cancel add_left_comm add_left_eq_self

Oh no! On the way to `add_comm`, a wild `succ_add` appears. `succ_add a b` is the proof that `(succ a) + b = succ (a + b)` for `a` and `b` numbers. This result is what's standing in the way of `x + y = y + x`. Again we have the problem that we are adding `b` to things, so we need to use induction to split into the cases where `b = 0` and `b` is a successor.

You might want to think about whether induction on `a` or `b` is the best idea.

Theorem `succ_add`: For all natural numbers a, b, we have $\mathrm{succ}(a) + b = \mathrm{succ}(a + b)$.

Active Goal

Objects:

a b : ℕ

Goal:

succ a + b = succ (a + b)

```
induction b with b hb
```

Retry

Active Goal Goal 2

Objects:

a : ℕ

Goal:

succ a + 0 = succ (a + 0)

Execute

add_succ

`(a d : ℕ) : a + MyNat.succ d = MyNat.succ (a + d)`

`add_succ a b` is the proof of `a + succ b = succ (a + b)`.

Oh no! On the way to `add_comm`, a wild `succ_add` appears. `succ_add a b` is the proof that `(succ a) + b = succ (a + b)` for `a` and `b` numbers. This result is what's standing in the way of `x + y = y + x`. Again we have the problem that we are adding `b` to things, so we need to use induction to split into the cases where `b = 0` and `b` is a successor.

You might want to think about whether induction on `a` or `b` is the best idea.

Theorem `succ_add`: For all natural numbe
$\mathrm{succ}(a+b)$.

Active Goal

Objects:

a b : ℕ

Goal:

succ a + b = succ (a + b)

```
induction b with b hb
```

Active Goal | Goal 2

Objects:

a : ℕ

Goal:

succ a + 0 = succ (a + 0)

Theorems

* + 012 Peano ^ ≤

`a + b = b + a`

`succ n = n + 1`

`succ a + b = succ (a + b)`

`a + succ d = succ (a + d)`

`0 + n = n` `n + 0 = n`

Tactics

induction rw rfl

Definitions

\+ ℕ

Tactics

apply cases contrapose decide exact have induction intro left rfl right rw simp simp_add symm tauto trivial use

Definitions

* ^ + ≠ ≤ ℕ

Theorems

* + 012 Peano ^ ≤

add_comm add_succ add_zero succ_add succ_eq_add_one zero_add add_assoc add_left_cancel add_left_comm add_left_eq_self

Key ideas:

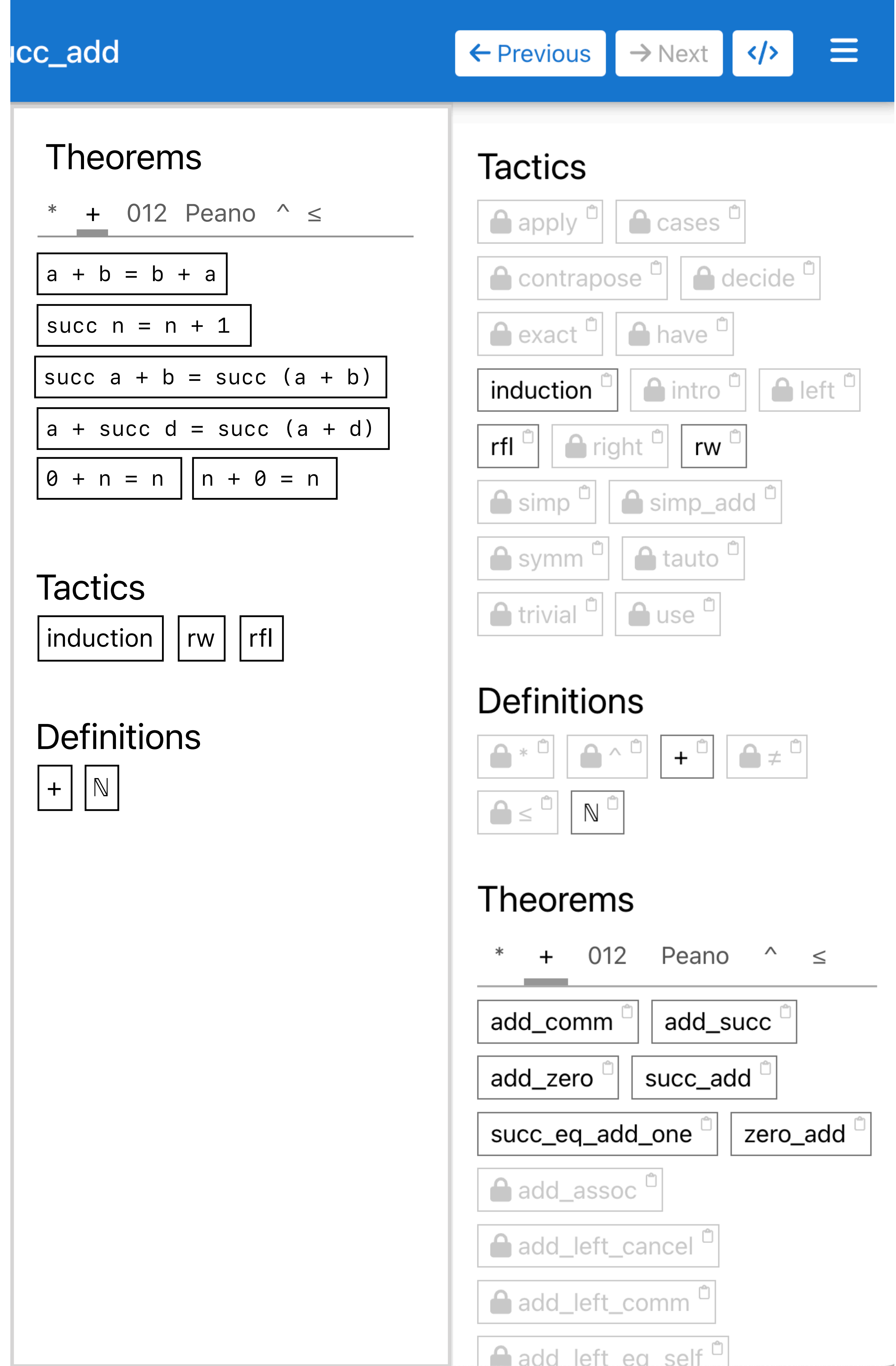
Memory vs. perception

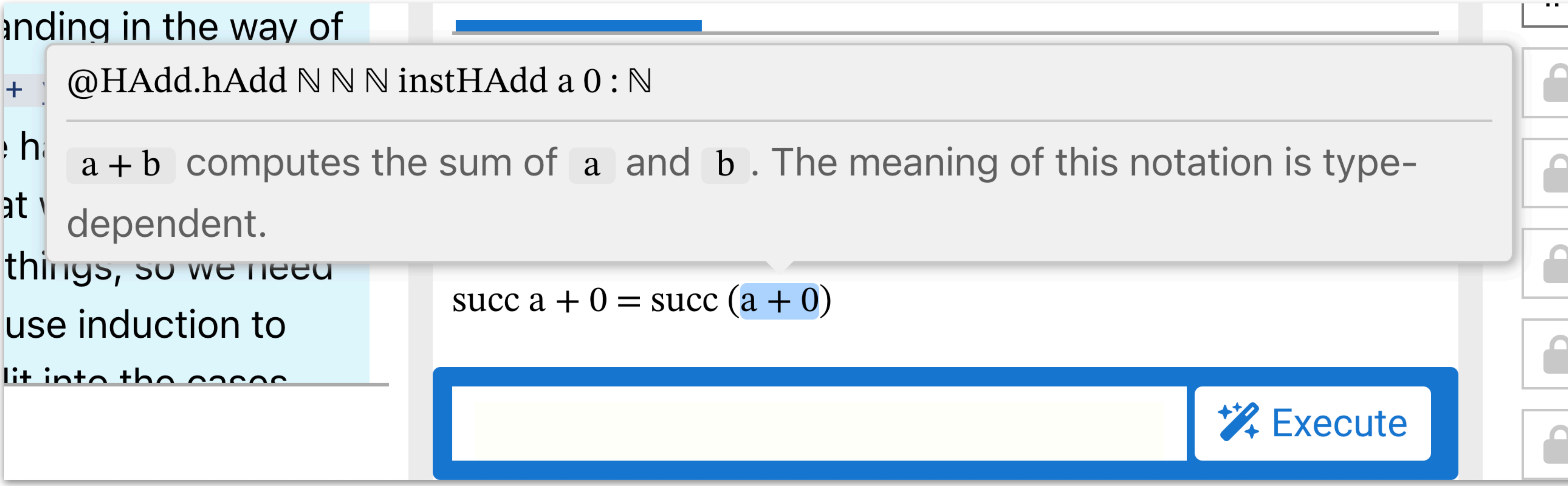
🧠 👀

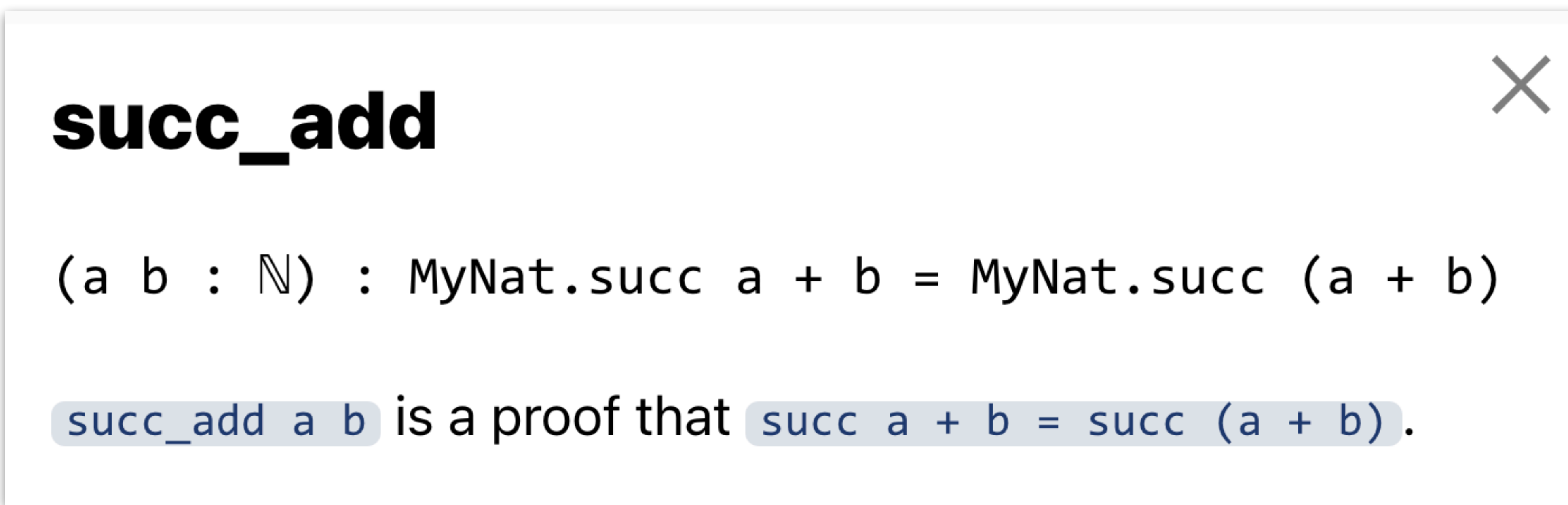
See: "The Role of Working Memory in Program Tracing"
"A Representational Analysis of Numeration Systems"

Cognitive load theory

See: John Sweller's publications







Language levels!

See: "The Structure and Interpretation of the Computer Science Curriculum"
How to Design Programs

“declarative”

Dependent types
Common theorems
Tactics

“procedural”

Decompose a problem
Find a theorem
Read the docs

What concepts and skills does a person need to effectively use Lean?

Undergrad math major
Professional C programmer
Experienced Coq dev

Model a domain
Prove a theorem
Debug a failure

1. Are learners actually learning?
2. What concepts/skills are missing?

Natural Number Game

Concepts

- Syntax
- Props & tactics
- Common tactics
- Basic arithmetic theorems
- Formalizable statements/proofs

Skills

- Generating syntax
- Selecting tactics
- Finding theorems
- Proof decomposition

Theorem Proving in Lean

Concepts

- Syntax
- Dependent types
- Term language
- Propositional logic
- Term vs. tactic proofs
- Forall/exists quantifiers
- Module system
- Inductive types & derived principle
- Mutual recursion
- Type classes and

Skills

- ?

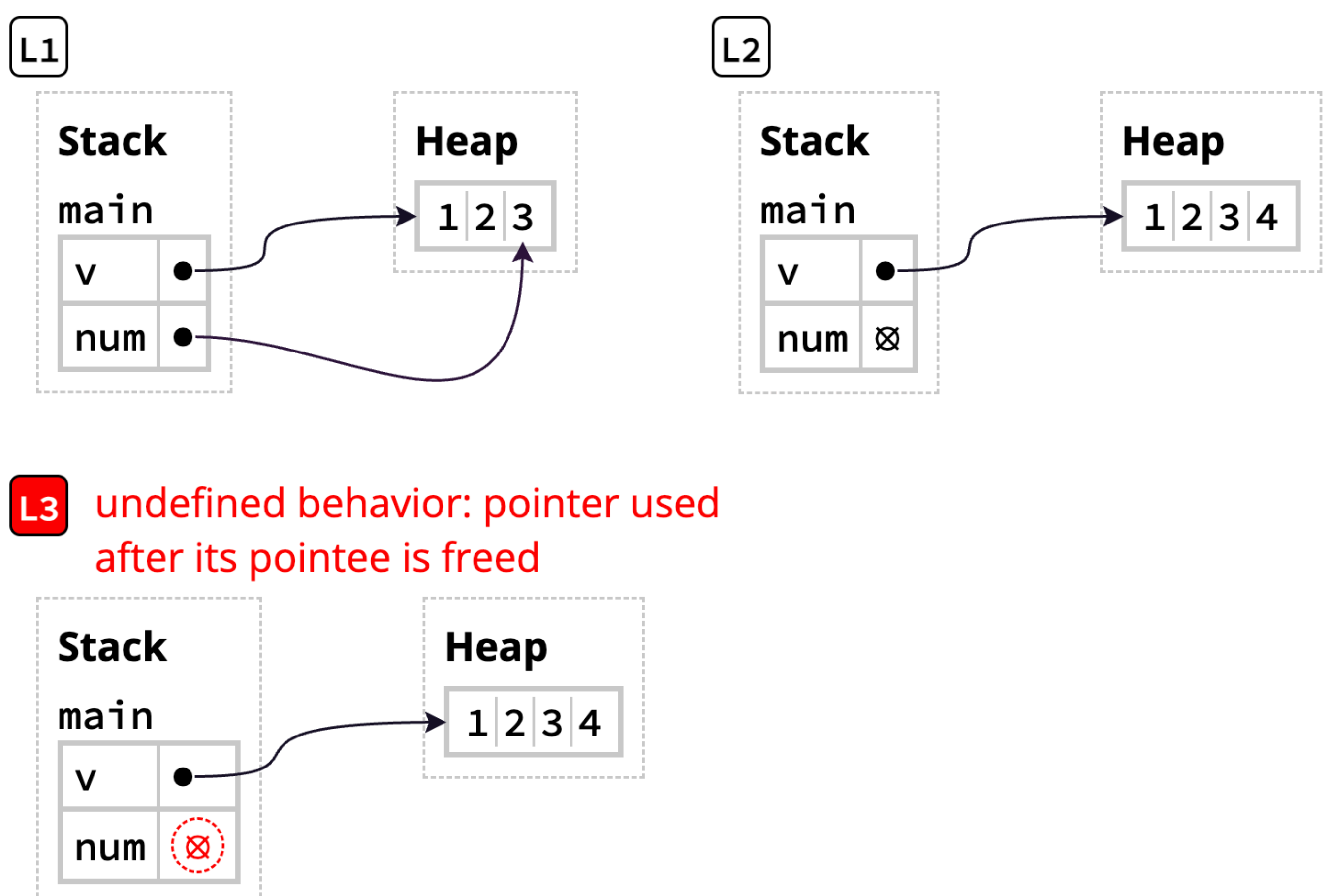
References and Borrowing - T

rust-book.cs.brown.edu/ch04-02-references-and-borrowing.html

The Rust Programming Language

To tie this back to memory safety, let's bring references into the mix. Say we created a reference to a vector's heap data. Then that reference can be invalidated by a push, as simulated below:

```
let mut v: Vec<i32> = vec![1, 2, 3];
let num: &i32 = &v[2]; L1
v.push(4); L2
println!("Third element is {}", *num); L3
```



https://rust-book.cs.brown.edu

called *lifetime parameters*. We will explain that feature later in Chapter 10.3, "Validating References with Lifetimes". For now, it's enough to know that: (1) input/output references are treated differently than references within a function body, and (2) Rust uses a different mechanism, the **F** permission, to check the safety of those references.

To see the **F** permission in another context, say you tried to return a reference to a variable on the stack like this:

```
fn return_a_string() -> &String {
    let s = String::from("Hello world");
    let s_ref = &s;
    R
    F s_ref
}
```

This program is unsafe because the reference `&s` will be invalidated when `return_a_string` returns. And Rust will reject this program with a similar `missing lifetime specifier` error. Now you can understand that error means that `s_ref` is missing the appropriate flow permissions.

Quiz 3 questions

Start

Quiz

Question 1 / 3

Question 1

Determine whether the program will pass the compiler. If it passes, write the expected output of the program if it were executed.

```
fn incr(n: &mut i32) {
  *n += 1;
}

fn main() {
  let mut n = 1;
  incr(&n);
  println!("{n}");
}
```

Response

This program: ○ DOES compile OR ○ does NOT compile

Submit

Question Summary

Quiz	Version	Question	Avg question score ▲ 0.1 0.3 0.4 0.6 0.8 1.0	Confidence Interval	N 56 10666 21276 31885 42495 53105
ch19-01-unsafe-rust	2	3	0.07	[0.05 - 0.08]	1373
ch11-02-running-tests	3	2	0.07	[0.06 - 0.07]	4090
ch04-01-ownership-sec2-mo…	8	4	0.07	[0.07 - 0.08]	8240
ch04-03-fixing-ownership-er…	5	4	0.09	[0.09 - 0.10]	5777
ch17-05-design-challenge-tr…	1	1	0.10	[0.08 - 0.12]	874
ch15-02-deref	2	1	0.11	[0.09 - 0.12]	1918
ch07-04-use	6	2	0.12	[0.11 - 0.13]	5403
ch16-01-threads	1	1	0.12	[0.11 - 0.13]	3182
ch06-04-inventory	4	5	0.15	[0.14 - 0.16]	7200
ch17-05-design-challenge-di…	1	1	0.16	[0.13 - 0.19]	755
ch17-04-inventory	3	4	0.17	[0.15 - 0.18]	1912
ch17-05-design-challenge-re…	1	3	0.17	[0.15 - 0.20]	1064

Quiz ch15-02-deref / Question 1

Question

Question 1

Determine whether the program will pass the compiler. If it passes, write the expected output of the program if it were executed. If the program does not pass, indicate the last line number involved in the compiler error.

```
1  use std::ops::Deref;
2
3  #[derive(Clone, Copy)]
4  struct AccessLogger(i32);
5
6  impl Deref for AccessLogger {
7      type Target = i32;
8      fn deref(&self) -> &Self::Target {
9          println!("deref");
10         &self.0
11     }
```

Answers

N = 766

This program **does not** compile.

The last line number in the error is:

N = 548

This program **does** compile.

The output of this program will be:

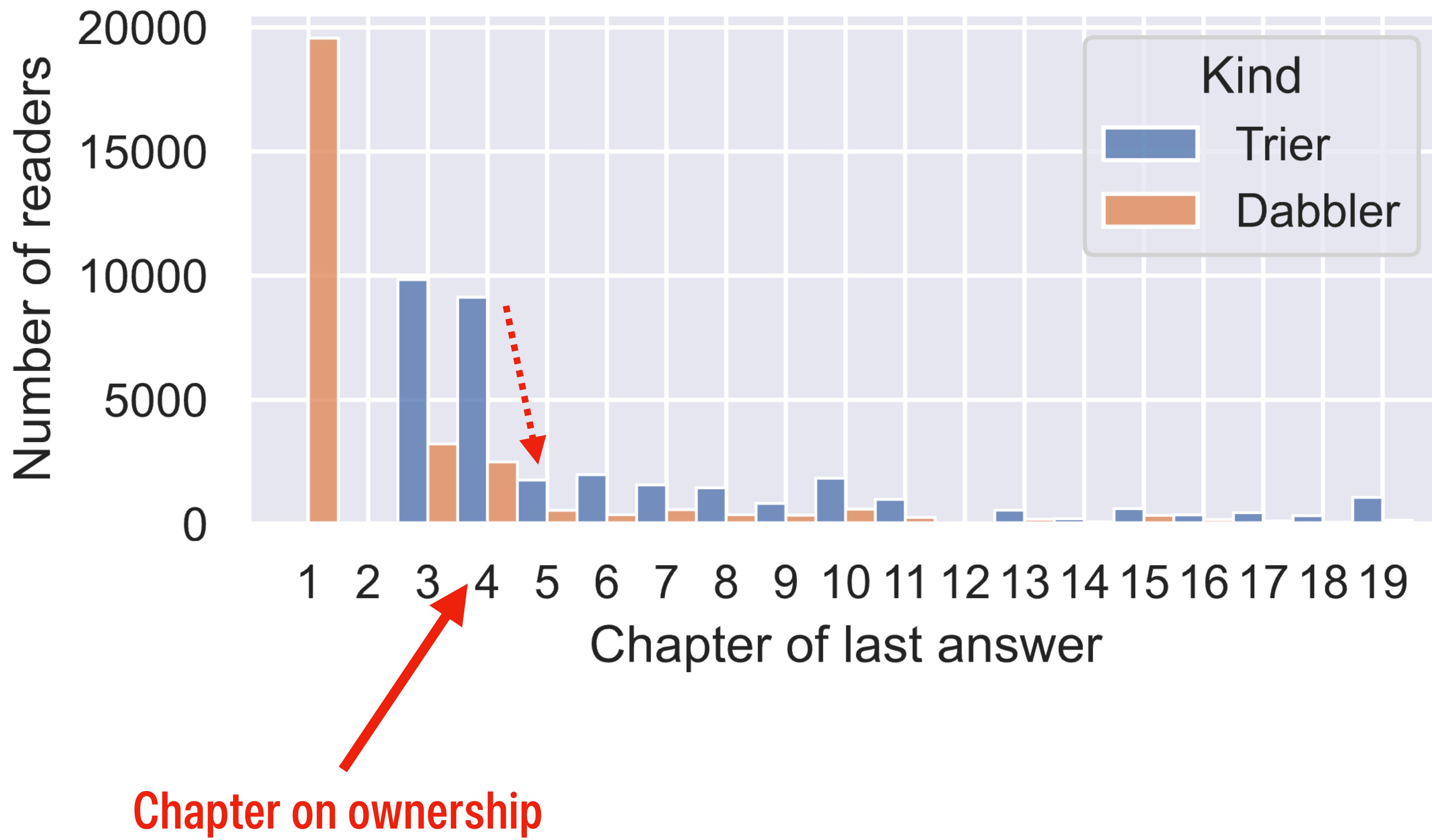
```
0 -1
```

Table 1. Effects of interventions on question accuracy. p-values are bolded if they are under the 0.05 significance threshold. p-values are corrected for multiple comparisons with the Benjamini–Hochberg method [1995].

Description	**Before**	***N***	**After**	***N***	**Δ**	***d***	***p***
Semver dependency deduplication	0.18	593	0.70	543	0.52	1.24	**<0.001**
Rust lacks inheritance	0.29	234	0.74	3511	0.45	1.03	**<0.001**
Match expressions and ownership	0.39	522	0.74	4970	0.35	0.78	**<0.001**
Send vs. Sync	0.25	639	0.49	538	0.24	0.52	**<0.001**
String slice diagram	0.23	575	0.43	7188	0.20	0.41	**<0.001**
Heap allocation with strings	0.13	265	0.27	3636	0.14	0.32	**<0.001**
Rules of lifetime inference	0.26	177	0.40	2887	0.14	0.29	**<0.001**
Traits vs. templates	0.38	234	0.49	3511	0.11	0.21	**0.002**
Trait objects and type inference	0.09	654	0.18	544	0.09	0.27	**<0.001**
Refutable patterns	0.17	549	0.25	499	0.08	0.21	**0.001**
Declarative macros take items	0.19	340	0.23	312	0.04	0.09	0.253
Derefencing vector elements	0.15	311	0.18	4001	0.03	0.07	0.253
				Grand average:	**0.20**	**0.45**	

Anyone can do this!

Crichton and Krishnamurthi. “Profiling Programming Language Learning” OOPSLA ‘24



Crichton and Krishnamurthi. "Profiling Programming Language Learning" OOPSLA '24

Steps to improving ownership pedagogy

1. Collect frequent StackOverflow questions about ownership
2. Ask Rust learners to solve these questions
3. Qualitatively analyze their misconceptions
4. Develop new materials to address the specific misconceptions
5. Deploy the materials in the online textbook
6. Measure effect on those same questions

Anyone can do this (with enough effort)!

Crichton, Gray, and Krishnamurthi. “A Grounded Conceptual Model for Ownership Types in Rust” OOPSLA ‘23 new members > Understanding newbie questions AUG 7

Will Crichton 1:40 PM

Hi folks, I am giving a Topos Institute talk in a few weeks about the human factors of formalized mathematics. I'm gathering some anecdata about the most common challenges faced by Lean newbies. So if you frequently answer questions in #new members, a few questions for you:

1. What are the most common categories of questions asked by newbies? Does it differ between mathematicians and software engineers?
2. Which Lean features do newbies struggle with the most early on? Which Lean features does everyone seem to struggle with no matter how long they've used Lean?
3. What are common misconceptions about formalized mathematics that you observe in newbie questions? What resources do you use to correct those misconceptions?

Feel free to answer any or all of these, and thank you in advance!

Using Lean

```lean
example (p q r : Prop)
  : p ∧ (q ∨ r) ↔ (p ∧ q) ∨ (p ∧ r) := by
  apply Iff.intro
  · intro h
    cases h.right with
    | inl hq =>
      show (p ∧ q) ∨ (p ∧ r)
      exact Or.inl ⟨h.left, hq⟩
    | inr hr =>
      show (p ∧ q) ∨ (p ∧ r)
      exact Or.inr ⟨h.left, hr⟩
  · intro h
    cases h with
    | inl hpq =>
      show p ∧ (q ∨ r)
      exact ⟨hpq.left, Or.inl hpq.right⟩
    | inr hpr =>
      show p ∧ (q ∨ r)
      exact ⟨hpr.left, Or.inr hpr.right⟩
```

▼Tactic state

1 goal

```
▼case mp
p q r : Prop
h : p ∧ (q ∨ r)
x✝ : q ∨ r
⊢ p ∧ q ∨ p ∧ r
```

▶ All Messages (0)

At a given point in a proof...

What facts have been locally created?

What am I trying to prove right now?

```
section LayoutMonotone
theorem layoutFluidAt.monotone (dim : Fin 2) (els : List FluidEl) (p : Pos)
        apply Rat.add_nonneg el1.el.box.size.2.prop
        exact h_valid.left

    case inr h_el' =>
      have h_suffix : els.IsStrictSuffix (el1 :: el2 :: els) := by
        existsi [el1, el2]
        constructor <;> simp

      cases els with
      | nil => trivial
      | cons el3 els =>
        -- TODO: rewrite this using layoutFormula
        have := ih (el3 :: els) h_suffix
          (p + el1.prevMargin.toVec dim
            + el1.el.box.size.nthVec dim + el1.nextMargin.toVec dim + el2.prevMargin.toV
            + el2.el.box.size.nthVec dim + el2.nextMargin.toVec dim)
          (List.Chain'.tail h_valid.right)
          h_el'
        apply le_trans _ this; simp

        have h_el1_el2 := h_valid.left
        have := List.chain'_cons.mp h_valid.right
        have h_el2_el3 := this.left
        simp [valid_adjacent_fluid_els] at h_el1_el2 h_el2_el3

        -- TODO: try and abstract out this logic
        cases dim.two_eq_or
        case inl h_dim =>
          simp [h_dim, Rat.toVec, Prod.nthVec, Prod.nth]
          rw [add_assoc, add_assoc, add_assoc, add_assoc, add_assoc]
          rw [le_add_iff_nonneg_right (p.1 + el1.prevMargin)]
          apply Rat.add_nonneg el1.el.box.size.1.prop
          rw [←add_assoc]; apply Rat.add_nonneg h_el1_el2
          apply Rat.add_nonneg el2.el.box.size.1.prop
          exact h_el2_el3
        case inr h_dim =>
          simp [h_dim, Rat.toVec, Prod.nthVec, Prod.nth]
          rw [add_assoc, add_assoc, add_assoc, add_assoc, add_assoc]
          rw [le_add_iff_nonneg_right (p.2 + el1.prevMargin)]
          apply Rat.add_nonneg el1.el.box.size.2.prop
          rw [←add_assoc]; apply Rat.add_nonneg h_el1_el2
          apply Rat.add_nonneg el2.el.box.size.2.prop
          exact h_el2_el3
```

1 goal

```
el' : Element
dim : Fin 2
el1 : FluidEl
p : Pos
el2 el3 : FluidEl
els : List FluidEl
ih : ∀ (l' : List FluidEl),
  l'.IsStrictSuffix (el1 :: el2 :: el3 :: els) →
    ∀ (p : Pos),
      valid_fluid_els l' →
        ∀ (h_el' : el' ∈ layoutFluidAt dim l' p), Prod.nth p dim + (l'.head ⋯).prevMargin ≤
Prod.nth el'.box.lo dim
h_valid† : valid_fluid_els (el1 :: el2 :: el3 :: els)
h_el'† : el' ∈ layoutFluidAt dim (el1 :: el2 :: el3 :: els) p
h_valid : valid_adjacent_fluid_els el1 el2 ∧ List.Chain' valid_adjacent_fluid_els (el2 :: el3
:: els)
h_el' : el' ∈
  layoutFluidAt dim (el3 :: els)
    (p + el1.prevMargin.toVec dim + (↑(Prod.nthVec el1.el.box.size dim).1, ↑(Prod.nthVec
el1.el.box.size dim).2) +
            el1.nextMargin.toVec dim +
          el2.prevMargin.toVec dim +
        (↑(Prod.nthVec el2.el.box.size dim).1, ↑(Prod.nthVec el2.el.box.size dim).2) +
      el2.nextMargin.toVec dim)
h_suffix : (el3 :: els).IsStrictSuffix (el1 :: el2 :: el3 :: els)
this† : Prod.nth
      (p + el1.prevMargin.toVec dim + (↑(Prod.nthVec el1.el.box.size dim).1, ↑(Prod.nthVec
el1.el.box.size dim).2) +
              el1.nextMargin.toVec dim +
            el2.prevMargin.toVec dim +
          (↑(Prod.nthVec el2.el.box.size dim).1, ↑(Prod.nthVec el2.el.box.size dim).2) +
        el2.nextMargin.toVec dim)
      dim +
    ((el3 :: els).head ⋯).prevMargin ≤
  Prod.nth el'.box.lo dim
h_el1_el2 : 0 ≤ el1.nextMargin + el2.prevMargin
this : valid_adjacent_fluid_els el2 el3 ∧ List.Chain' valid_adjacent_fluid_els (el3 :: els)
h_el2_el3 : 0 ≤ el2.nextMargin + el3.prevMargin
h_dim : dim = 0
⊢ p.1 + el1.prevMargin ≤
  p.1 + el1.prevMargin + ↑el1.el.box.size.1 + el1.nextMargin + el2.prevMargin +
↑el2.el.box.size.1 + el2.nextMargin +
    el3.prevMargin
```

▶ All Messages (0)

Restart File

Group related info

```
Prod.nth el'.box.lo dim
h_validt : valid_fluid_els (el1 :: el2 :: el3 :: els)
h_el't : el' ∈ layoutFluidAt dim (el1 :: el2 :: el3 :: els) p
h_valid : valid_adjacent_fluid_els el1 el2 ∧ List.Chain' valid_adjacent_fluid_els (el2 :: el3
:: els)
h_el' : el' ∈
  layoutFluidAt dim (el3 :: els)
    (p + el1.prevMargin.toVec dim + (↑(Prod.nthVec el1.el.box.size dim).1, ↑(Prod.nthVec
el1.el.box.size dim).2) +
            el1.nextMargin.toVec dim +
          el2.prevMargin.toVec dim +
        (↑(Prod.nthVec el2.el.box.size dim).1, ↑(Prod.nthVec el2.el.box.size dim).2) +
      el2.nextMargin.toVec dim)
h_suffix : (el3 :: els).IsStrictSuffix (el1 :: el2 :: el3 :: els)
thist : Prod.nth
      (p + el1.prevMargin.toVec dim + (↑(Prod.nthVec el1.el.box.size dim).1, ↑(Prod.nthVec
el1.el.box.size dim).2) +
              el1.nextMargin.toVec dim +
            el2.prevMargin.toVec dim +
          (↑(Prod.nthVec el2.el.box.size dim).1, ↑(Prod.nthVec el2.el.box.size dim).2) +
        el2.nextMargin.toVec dim)
      dim +
    ((el3 :: els).head ⋯).prevMargin ≤
  Prod.nth el'.box.lo dim
h_el1_el2 : 0 ≤ el1.nextMargin + el2.prevMargin
this : valid_adjacent_fluid_els el2 el3 ∧ List.Chain' valid_adjacent_fluid_els (el3 :: els)
h_el2_el3 : 0 ≤ el2.nextMargin + el3.prevMargin
h_dim : dim = 0
⊢ p.1 + el1.prevMargin ≤
  p.1 + el1.prevMargin + ↑el1.el.box.size.1 + el1.nextMargin + el2.prevMargin +
↑el2.el.box.size.1 + el2.nextMargin +
    el3.prevMargin
```

Abstract repeated details

Restart File

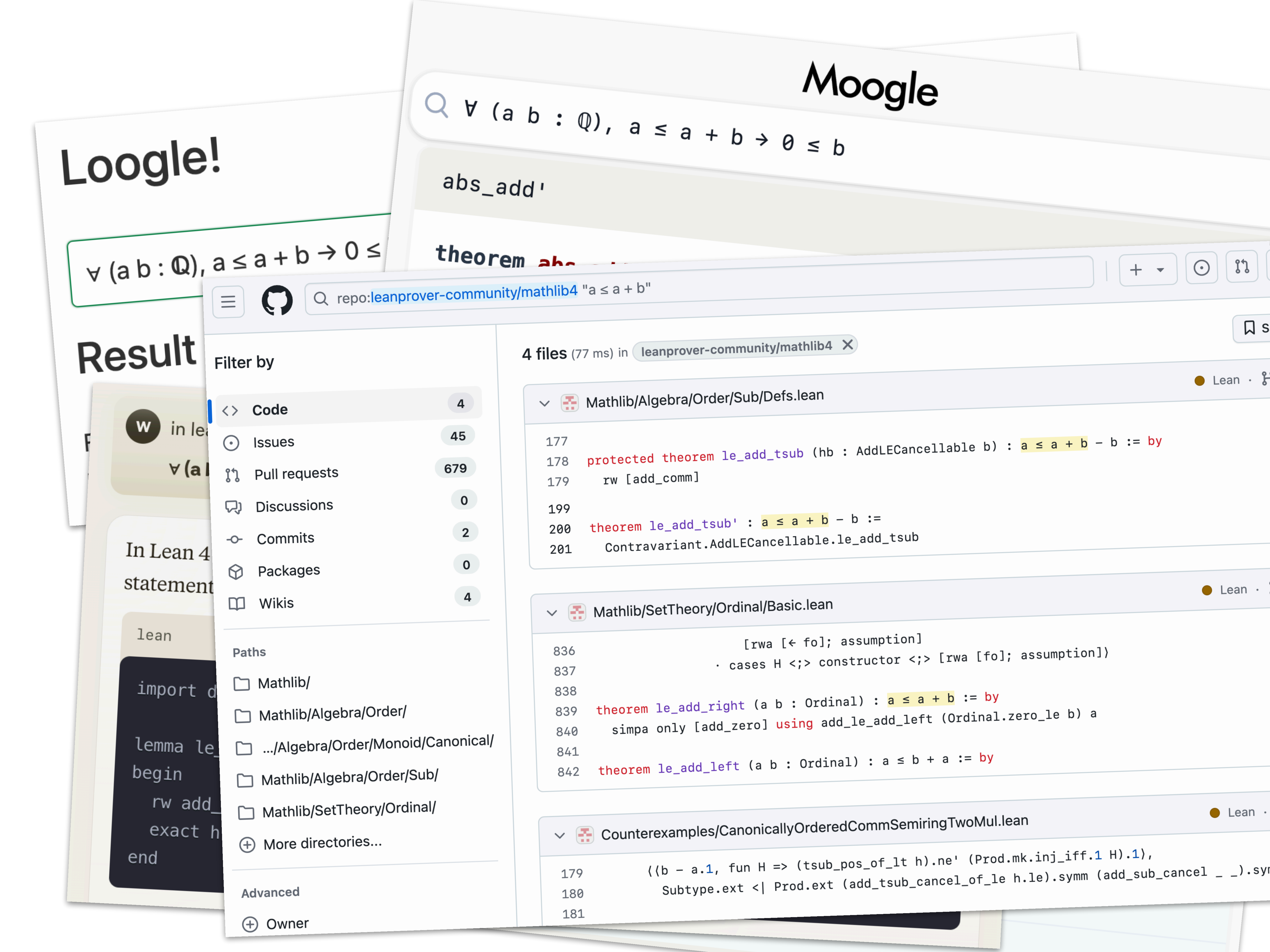
```
⊢ p.1 + el1.prevMargin ≤
  p.1 + el1.prevMargin +
    (↑el1.el.box.size.1 +
      (el1.nextMargin + (el2.prevMargin + (↑el2.el.box.size.1 +
(el2.nextMargin + el3.prevMargin)))))
```

“How can I get rid of `p.1 + el1.prevMargin` from both sides?”

```
apply?
```

```
timeout at `whnf`, maximum
number of heartbeats (200000)
```

```
∀ a b : ℚ, a ≤ a + b → 0 ≤ b
```


Loogle!
∀ (a b : ℚ), a ≤ a + b → 0 ≤
Result
Moogle
∀ (a b : ℚ), a ≤ a + b → 0 ≤ b
abs_add'
theorem
repo:leanprover-community/mathlib4 "a ≤ a + b"
Filter by
Code 4
Issues 45
Pull requests 679
Discussions 0
Commits 2
Packages 0
Wikis 4
Paths
Mathlib/
Mathlib/Algebra/Order/
.../Algebra/Order/Monoid/Canonical/
Mathlib/Algebra/Order/Sub/
Mathlib/SetTheory/Ordinal/
More directories...
Advanced
Owner
4 files (77 ms) in leanprover-community/mathlib4
Mathlib/Algebra/Order/Sub/Defs.lean
Lean
177
178 protected theorem le_add_tsub (hb : AddLECancellable b) : a ≤ a + b − b := by
179 rw [add_comm]
199
200 theorem le_add_tsub' : a ≤ a + b − b :=
201 Contravariant.AddLECancellable.le_add_tsub
Mathlib/SetTheory/Ordinal/Basic.lean
Lean
836 [rwa [← fo]; assumption]
837 · cases H <;> constructor <;> [rwa [fo]; assumption])
838
839 theorem le_add_right (a b : Ordinal) : a ≤ a + b := by
840 simpa only [add_zero] using add_le_add_left (Ordinal.zero_le b) a
841
842 theorem le_add_left (a b : Ordinal) : a ≤ b + a := by
Counterexamples/CanonicallyOrderedCommSemiringTwoMul.lean
Lean
179 ((b − a.1, fun H => (tsub_pos_of_lt h).ne' (Prod.mk.inj_iff.1 H).1),
180 Subtype.ext <| Prod.ext (add_tsub_cancel_of_le h.le).symm (add_sub_cancel _ _).sy
181
in Lean 4
statement
lean
import
lemma le
begin
rw add
exact h
end

```
def foo {a b : ℚ} (h : a ≤ a + b) : 0 ≤ b := by
  exact?
```

▼Suggestions

Try this: `exact nonneg_of_le_add_right h`

```
theorem nonneg_of_le_add_right
        {α : Type u_1} [AddZeroClass α] [LE α]
        [
         ContravariantClass α α (fun (x x_1 : α) => x + x_1)
          fun (x x_1 : α) => x ≤ x_1
         ]
        {a : α} {b : α} (h : a ≤ a + b) :
    0 ≤ b
```

∀ (a b : ℚ), a ≤ a + b → 0 ≤ b

#find #lucky

Result

Found 0 definitions mentioning Rat.instOfNat, Rat, LE.le, Rat.instLE, instHAdd, Iff, HAdd.hAdd, Rat.instAdd and OfNat.ofNat. Of these, 0 match your pattern(s).

|- ?a ≤ ?a + ?b → 0 ≤ ?b

#find #lucky

Result

Found 1929 definitions mentioning LE.le, HAdd.hAdd and OfNat.ofNat. Of these, one matches your pattern(s).

- nonneg_of_le_add_right Mathlib.Algebra.Order.Monoid.Unbundled.Basic
  ```
  ∀ {α : Type u_1} [inst : AddZeroClass α] [inst_1 : LE α]
  [inst_2 : ContravariantClass α α (fun x x_1 ⇒ x + x_1) fun x
  x_1 ⇒ x ≤ x_1] {a b : α}, a ≤ a + b → 0 ≤ b
  ```

How do we help people find `nonneg_of_le_add_right`?

Loogle, Mathlib docs: integration + training

AI tools: make them… better??

Search tactics: facilitate writing MWE

```
def foo {a b : ℚ} (h : a ≤ a + b) : 0 ≤ b := by
  exact?
```

Calls to action

- Read up on psych research (the replicable kinds)
  - Cognitive psychology: memory, perception, mental models, logical reasoning
  - Educational psychology: cognitive load, skill acquisition, contrasting cases
  - Cognitive engineering: representations, user modeling, trade-offs
  - Conspicuously avoided: HCI, "intuitive" interfaces, 1-hour user studies, …
- Build the infrastructure for evaluating human factors
  - Textbook quizzes, Zulip questions are both potential data sources
  - Validated assessments of competency (see: "Force Concept Inventory")
  - Never underestimate the power of talking to people and watching them work
  - Conspicuously avoided: IDE telemetry, user surveys
- Improve the cognitive efficiency of devtools
  - Reduce the friction of discovery (incl. discovering the discovery tools)
  - Visualize, don't dump information (incl. docs, error messages, etc)
  - Conspicuously avoided: proof widgets, AI consultants