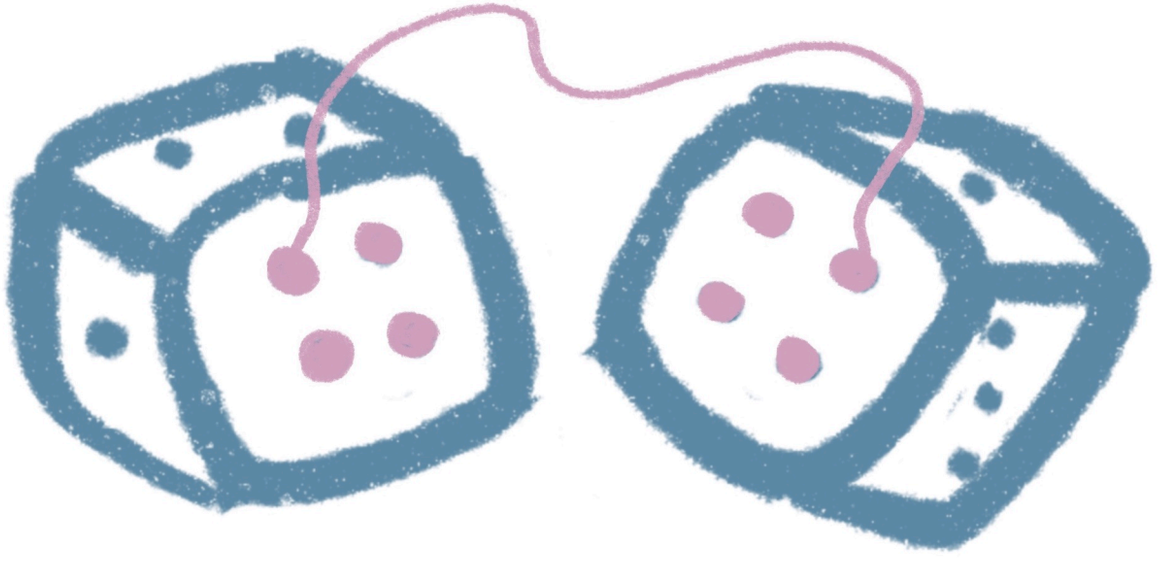
Automatically Generalizing Proofs Containing Linked Constants



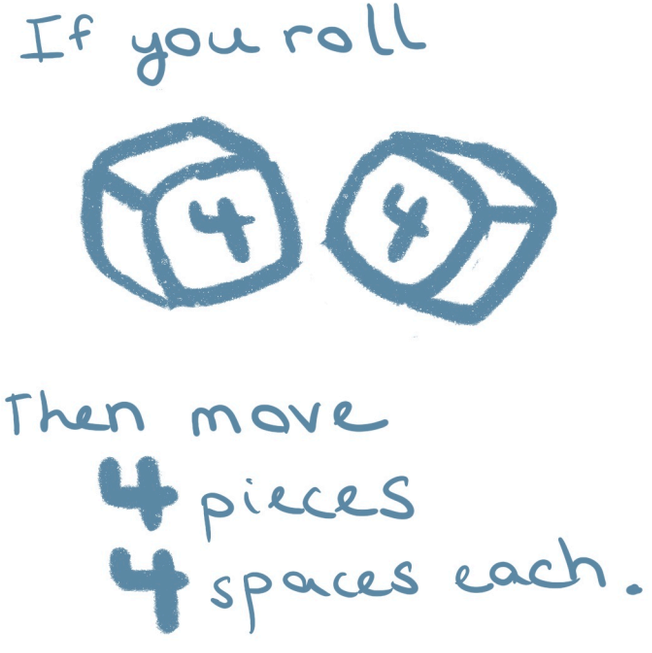
Anshula Gandhi | *PhD Candidate, University of Cambridge*

Anand Rao Tadipatri | *PhD Candidate, University of Cambridge*

Timothy Gowers | *Professor, University of Cambridge*

A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.



A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.

If you roll

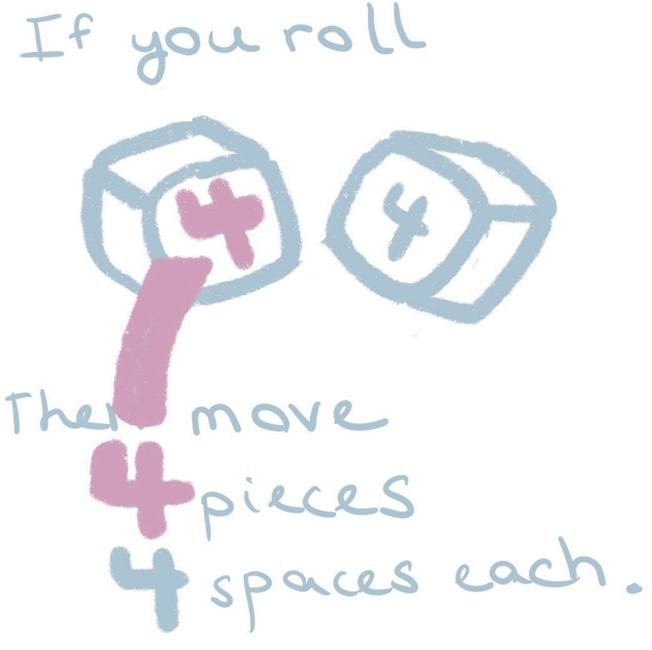
Then move
4 pieces
4 spaces each.

If you roll

... ? ...

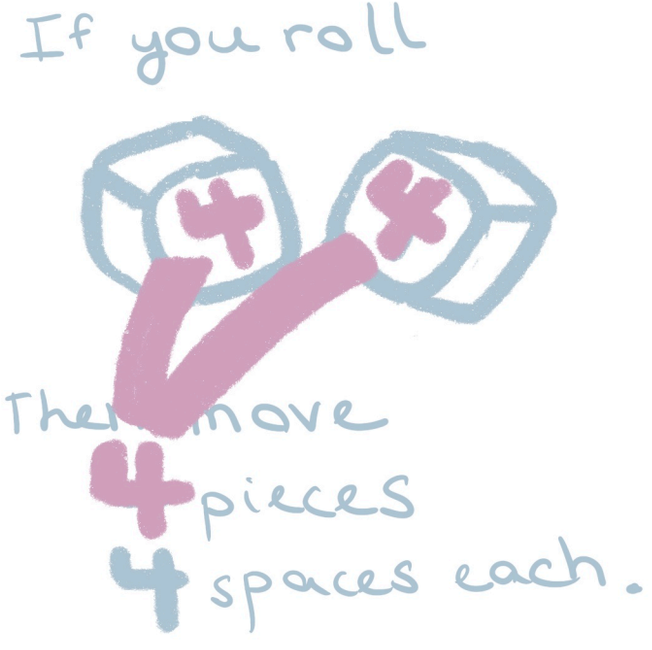
A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.



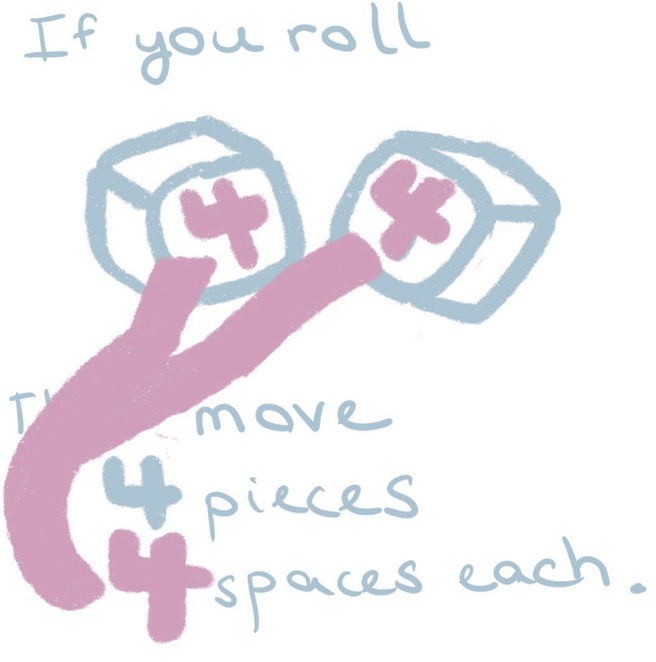
A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.



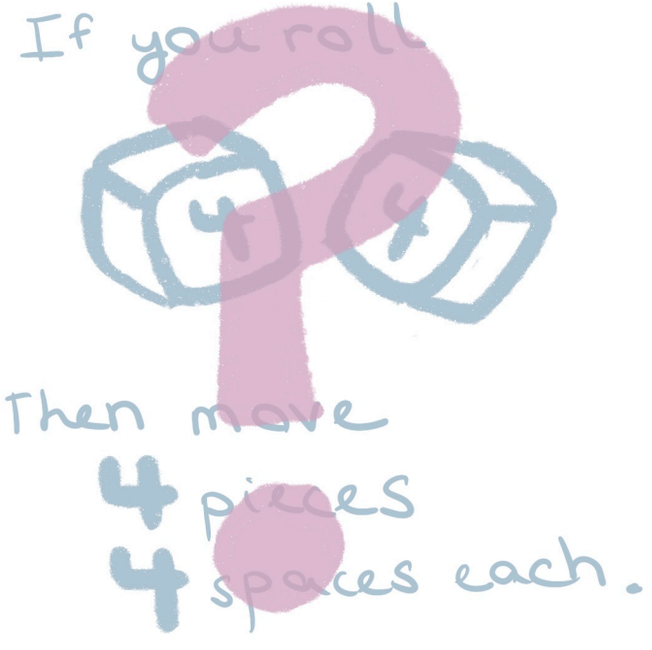
A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.



A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.

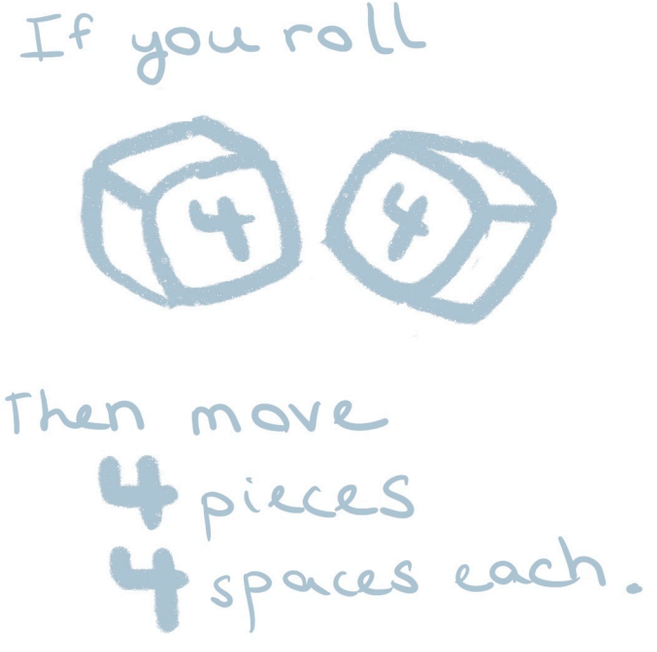


A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.

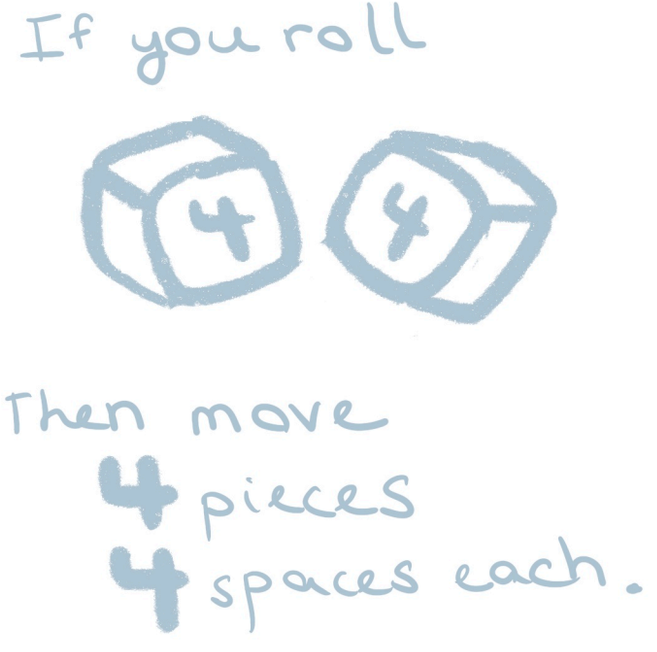
A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.



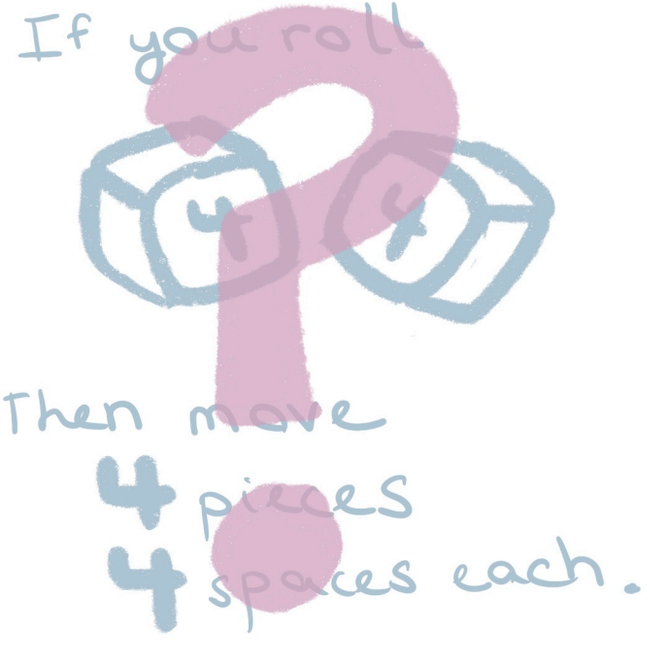
A Difficulty in Generalization

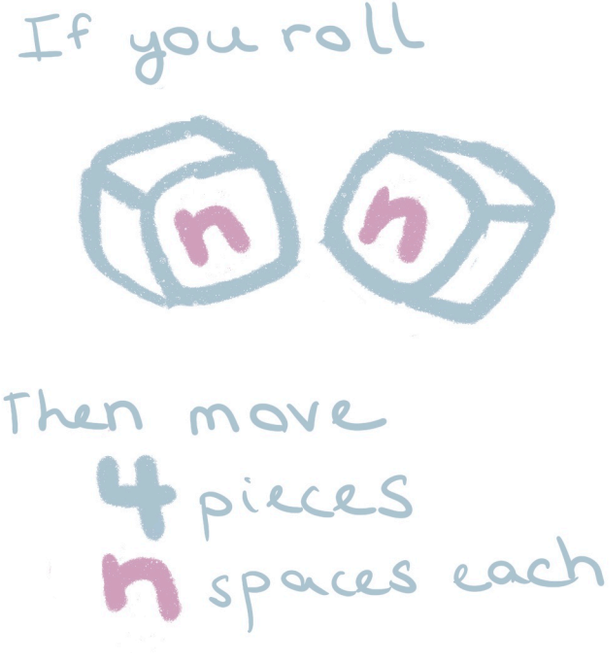
Generalization is more difficult when the expression you want to generalize appears multiple times.



A Difficulty in Generalization

Generalization is more difficult when the expression you want to generalize appears multiple times.





So often (but of course not always), the fewer repeated constants, the easier the generalization.

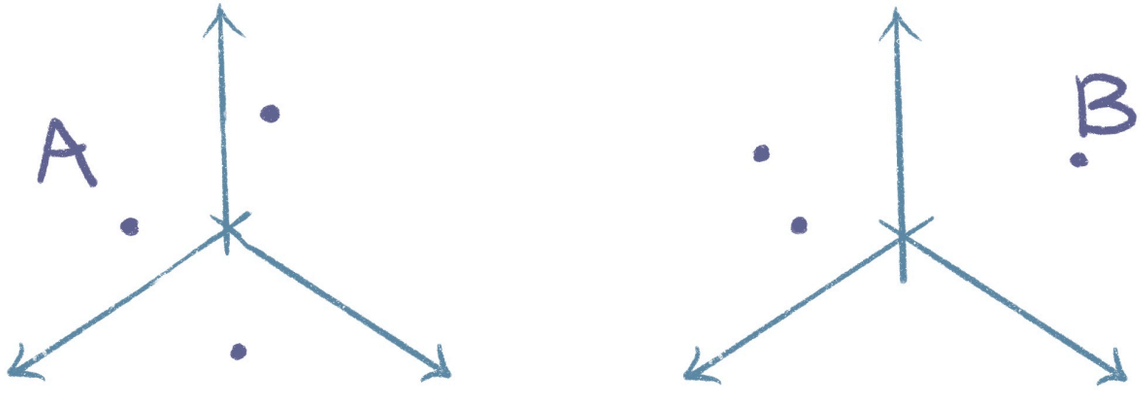
But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

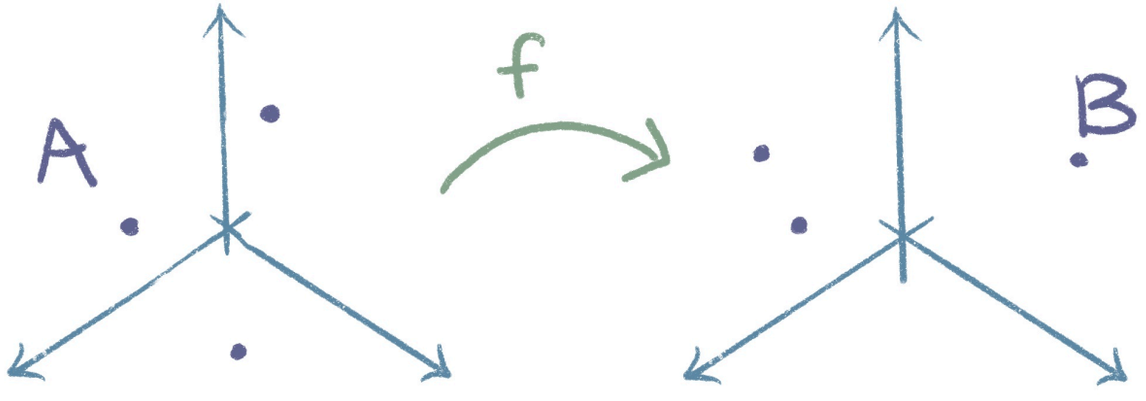
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

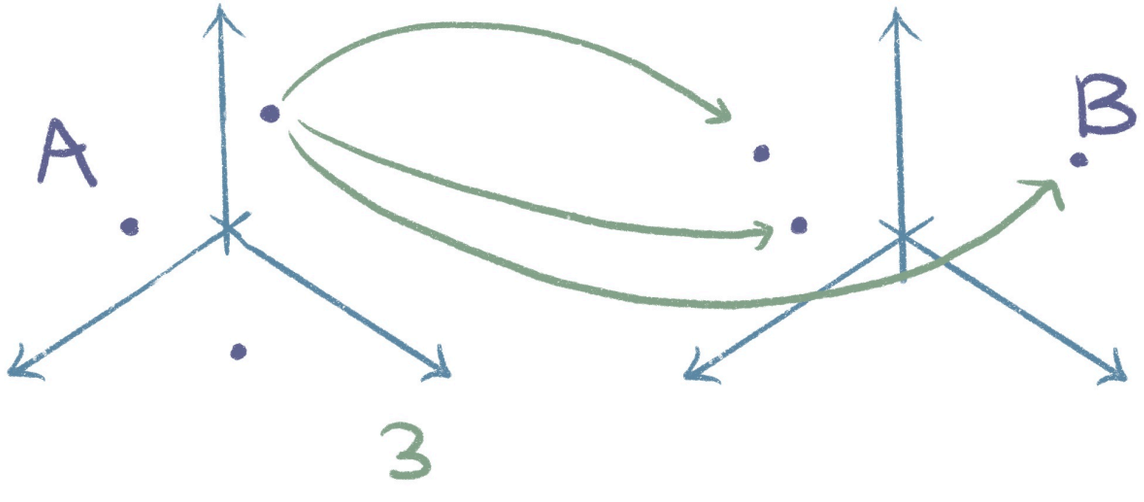
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

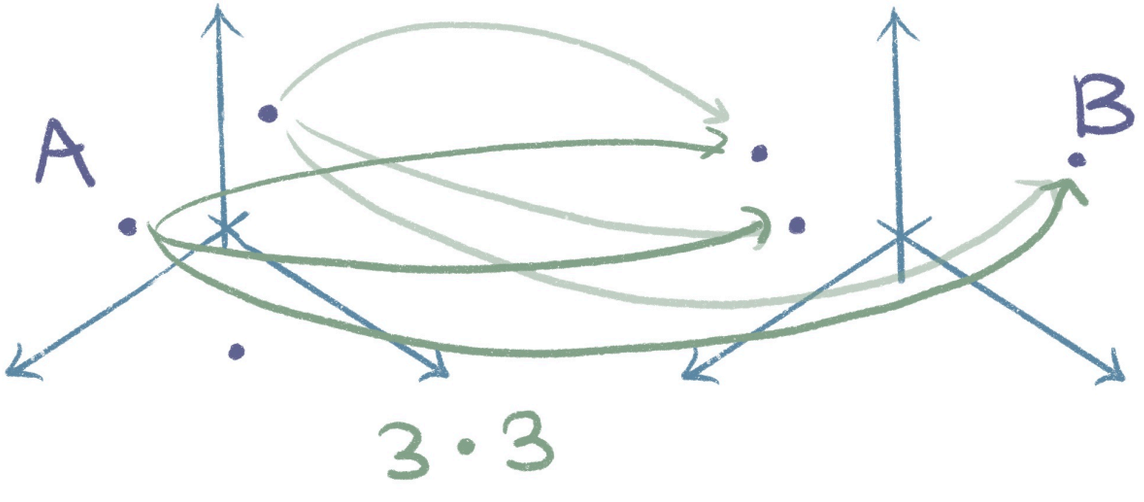
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

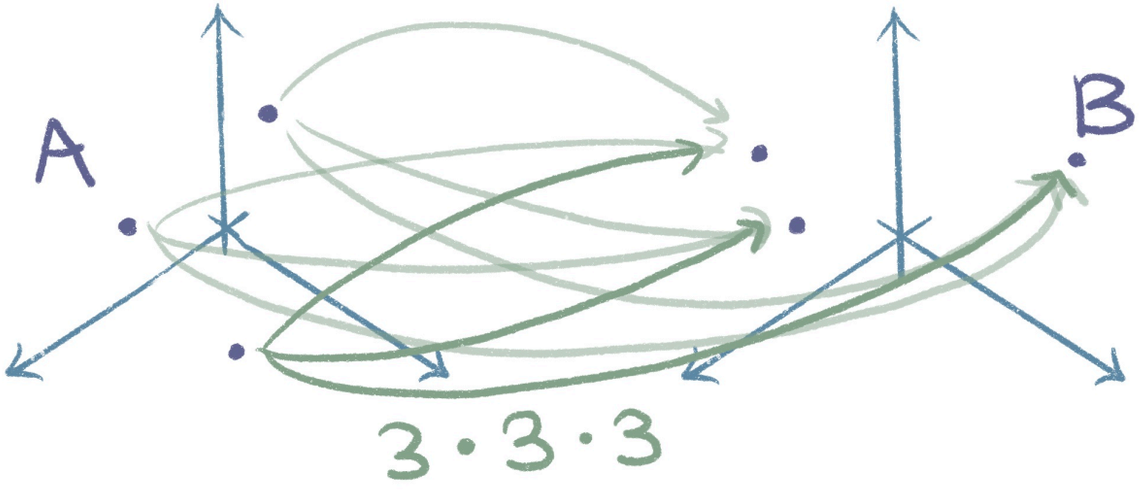
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

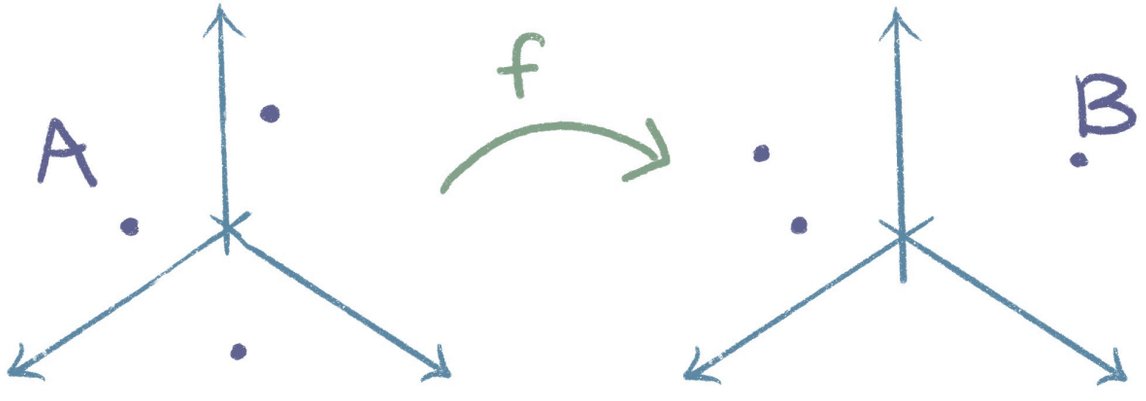
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

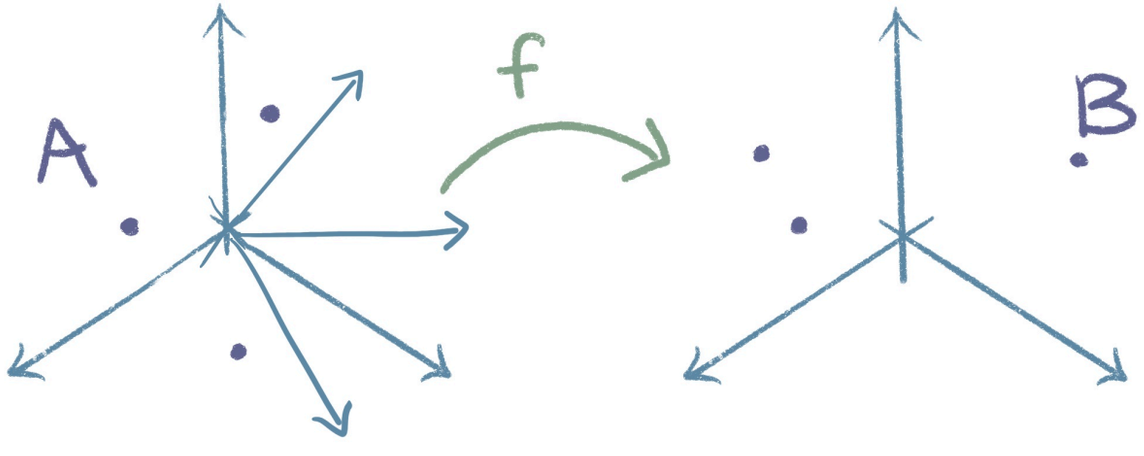
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

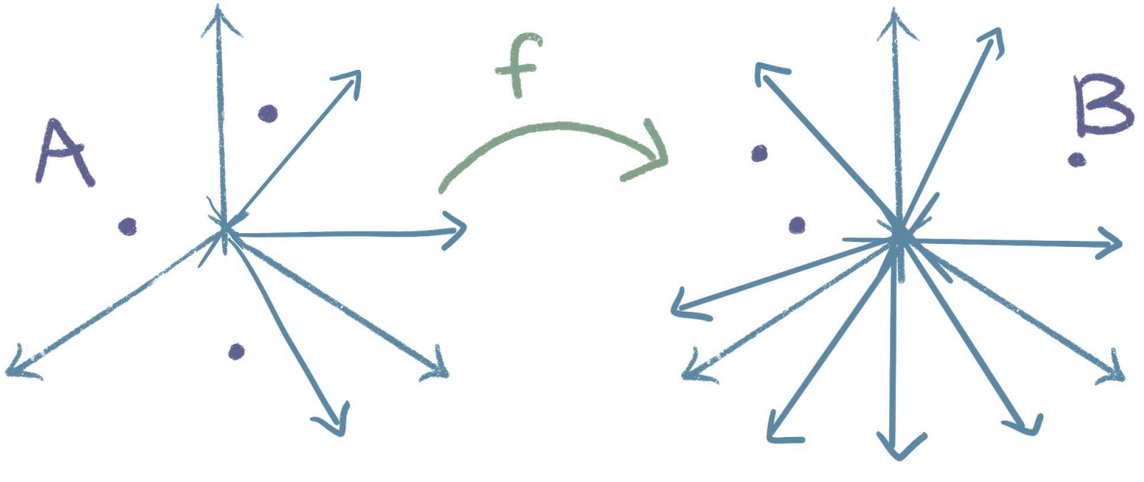
Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

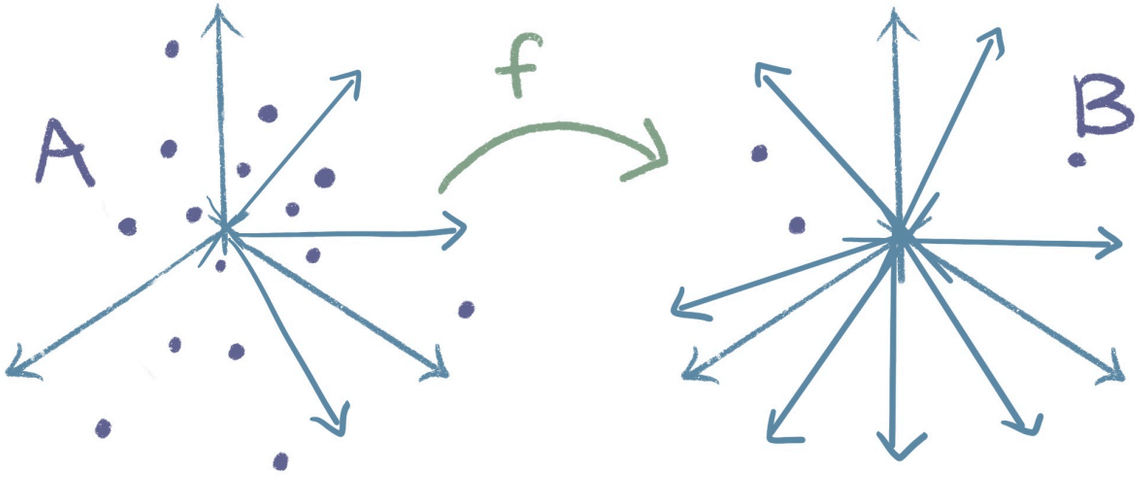
Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

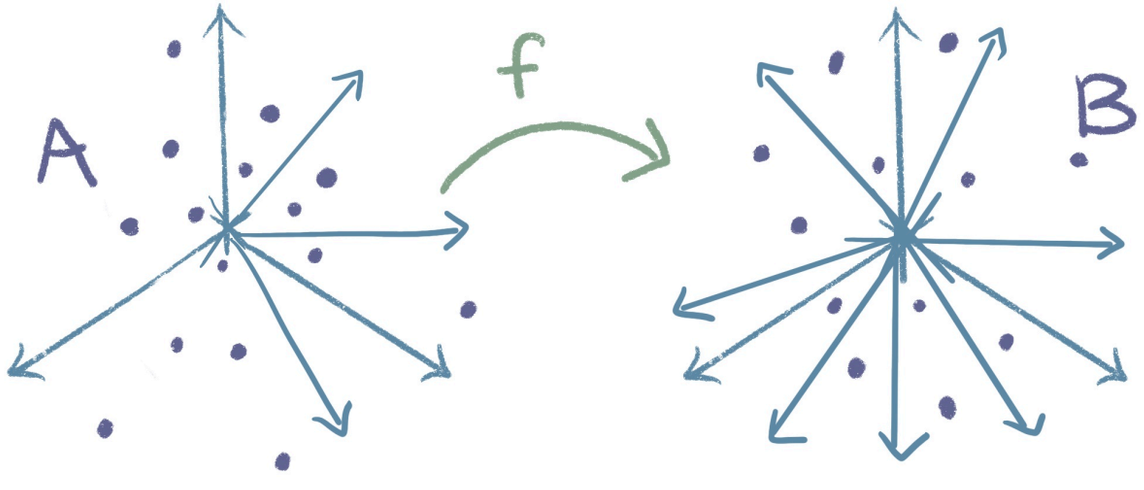
Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^a.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

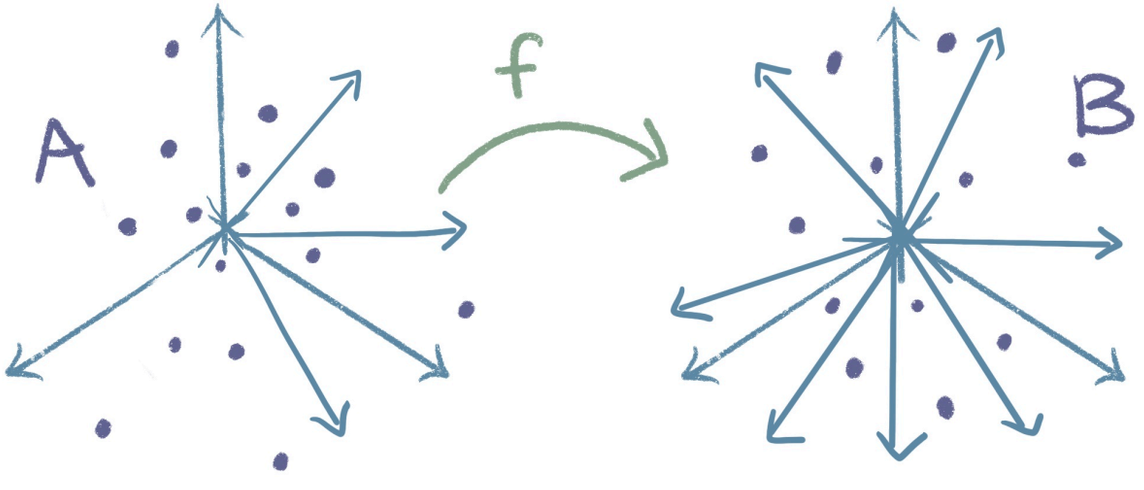
Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is b^a.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

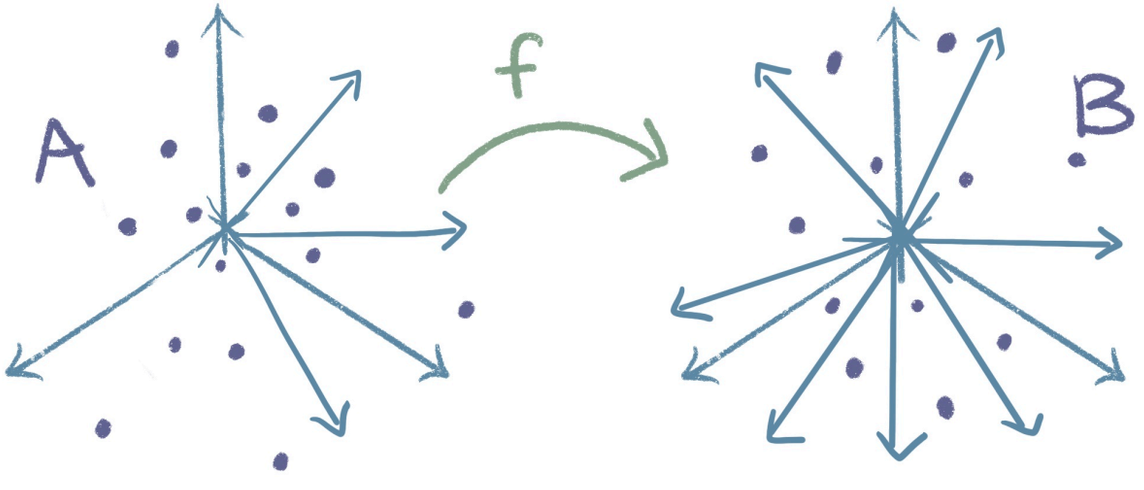
Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is b^a.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know*.

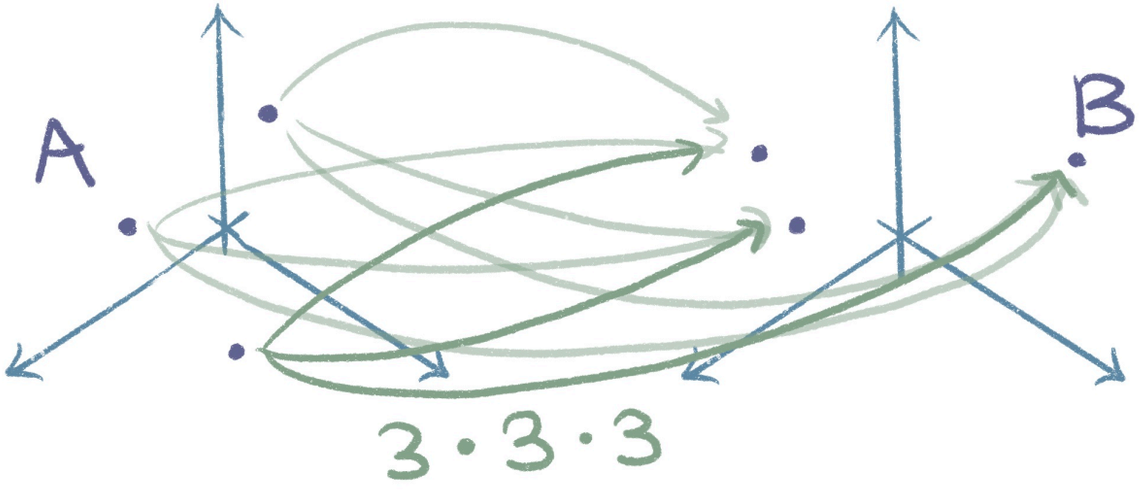
Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is b^a.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know* **from the proof**.

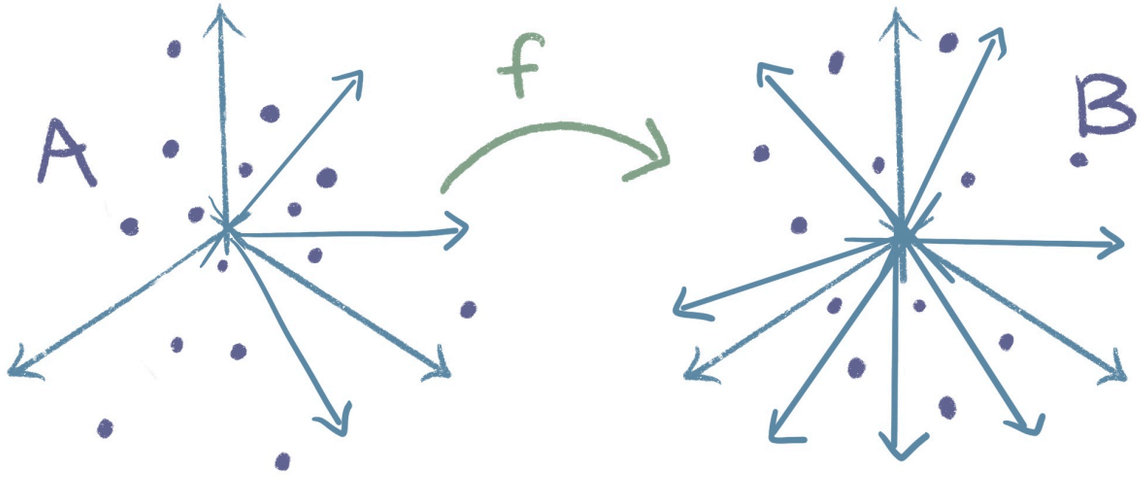
Theorem: Consider sets $A \subseteq \mathbb{R}^3$ and $B \subseteq \mathbb{R}^3$, where $|A| = 3$ and $|B| = 3$.
The number of functions $f : A \to B$ is 3^3.



But …mathematicians generalize statements with linked constants all the time.

Looking at a math proof, we don't need to *guess* which constants are linked. We *know* **from the proof**.

Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is b^a.

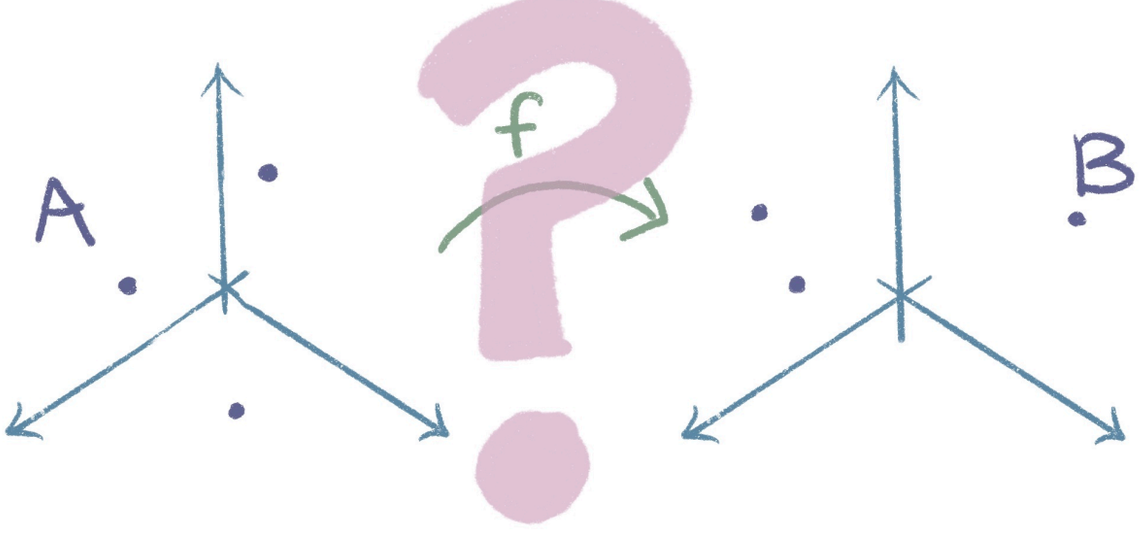


But …mathematicians generalize statements with linked constants all the time.

A mathematician equipped with the proof would never generalize…

- to something that isn't true (e.g. generalizing all "3"s arbitrarily)…

Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is m^n.

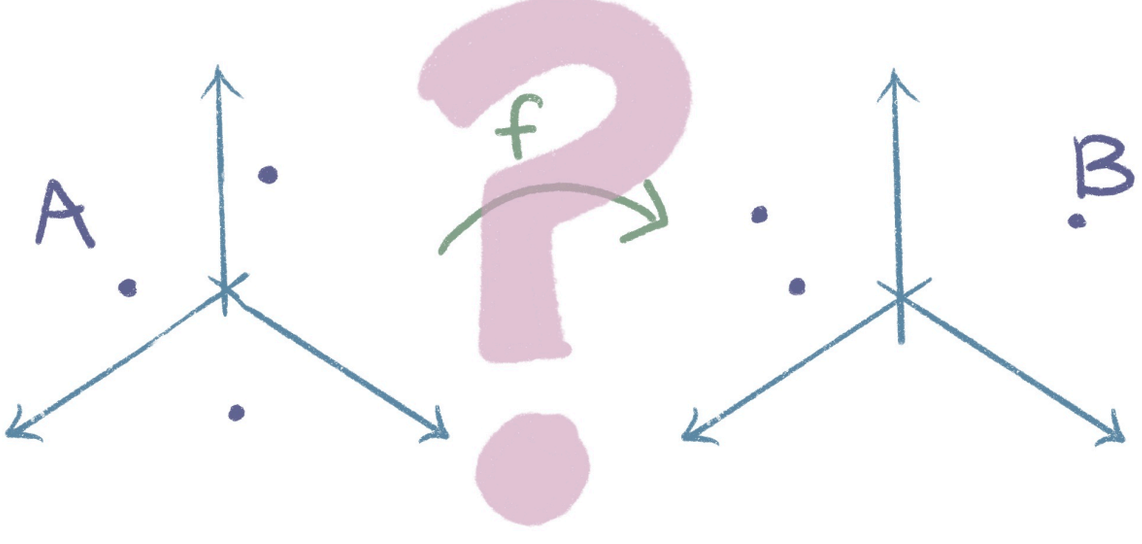


But …mathematicians generalize statements with linked constants all the time.

A mathematician equipped with the proof would never generalize…

- to something over-specific (generalizing all “3”s to the same “n”)…

Theorem: Consider sets $A \subseteq \mathbb{R}^n$ and $B \subseteq \mathbb{R}^n$, where $|A| = n$ and $|B| = n$.
The number of functions $f : A \to B$ is n^n.



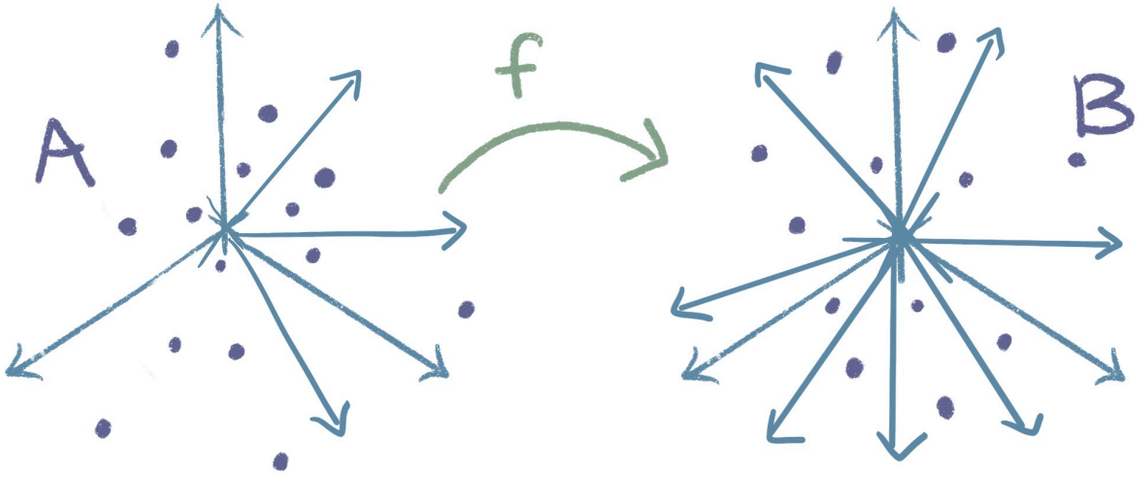
But …mathematicians generalize statements with linked constants all the time.

But without a proof, we might come up with generalizations that suffer from these suboptimalities.

But …mathematicians generalize statements with linked constants all the time.

How can we mechanize this generalization process, taking advantage of the proof?

Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is b^a.



How can we mechanize this generalization?

…without generalizing to something that isn't true?

- ?

…without generalizing to something overly specific?

- ?

How can we mechanize this generalization?

…without generalizing to something that isn't true (like below)?

Theorem: Consider sets $A \subseteq \mathbb{R}^m$ and $B \subseteq \mathbb{R}^n$, where $|A| = a$ and $|B| = b$.
The number of functions $f : A \to B$ is m^n.

…without generalizing to something overly specific?

- ?

How can we mechanize this generalization?

…without generalizing to something that isn't true?

- **(Already done)** An approach is suggested by "Generalization in type theory based proof assistants" by Olivier Pons.

…without generalizing to something overly specific ?

- ?

How can we mechanize this generalization?

…without generalizing to something that isn't true?

- **(Already done)** An approach is suggested by "Generalization in type theory based proof assistants" by Olivier Pons.

…without generalizing to something overly specific (like below)?

Theorem: Consider sets $A \subseteq \mathbb{R}^n$ and $B \subseteq \mathbb{R}^n$, where $|A| = n$ and $|B| = n$.
The number of functions $f : A \to B$ is n^n.

How can we mechanize this generalization?

…without generalizing to something that isn't true?

- **(Already done)** An approach is suggested by "Generalization in type theory based proof assistants" by Olivier Pons.

…without generalizing to something overly specific?

- **(Our proposal)** We suggest an approach (the contribution of this talk), involving unification of metavariables, which builds off the above approach by Pons.

How can we mechanize this generalization?

…without generalizing to something that isn't true? (problem #1)

- **(Already done)** An approach is suggested by "Generalization in type theory based proof assistants" by Olivier Pons.

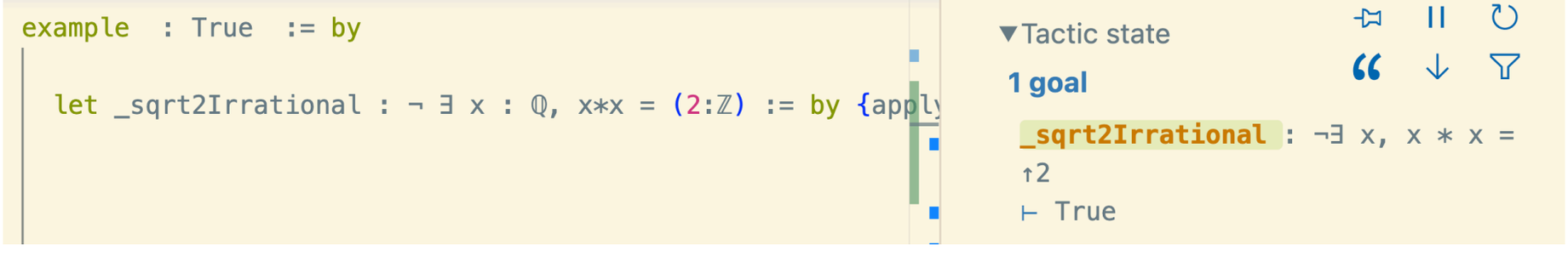
…without generalizing to something overly specific? (problem #2)

- **(Our proposal)** We suggest an approach (the contribution of this talk), involving unification of metavariables, which builds off the above approach by Pons.

The next part of this talk will delve into specifics of how we solve these two problems.

Problem #1: How do we avoid generalizing to something that isn't true?

The paper (Pons, 2000) proposes an algorithm for risk-free generalization of proofs.



Problem #1: How do we avoid generalizing to something that isn't true?

The paper (Pons, 2000) proposes an algorithm for risk-free generalization of proofs. We implemented it in Lean as a tactic: `autogeneralize_basic`.

Problem #1: How do we avoid generalizing to something that isn't true?

Note, however, that **the algorithm does not generate the theorem that the square root of any non-perfect-square is irrational** (which is also true) – because that is not evident in the proof term.

The algorithm doesn't determine *the most general* version of a statement – it determines the most general version of a statement *that the given proof allows*.

Problem #1: How do we avoid generalizing to something that isn't true?

So we start from the proof that 2 has an irrational square root:

```
_sqrt2Irrational : ¬∃ x, x * x = ↑2
```

And generalize to the proof that any prime has an irrational square root:

```
_sqrt2Irrational.Gen : ∀ (n : ℤ),
Prime n → ¬∃ x, x * x = ↑n
```

Pons's algorithm can be implemented in any type-theory-based theorem prover (it was initially implemented in Rocq, and we implemented it in Lean).

How does the algorithm work?

Problem #1: How do we avoid generalizing to something that isn't true?

Suppose we are given …

- a proof (e.g. that the square root of 2 is irrational).
- an expression to generalize in the proof (e.g. 2).

What do we do?

Problem #1: How do we avoid generalizing to something that isn't true?

1. Look at the proof.

   Why? To check which properties of "2" are actually used in the proof.

Problem #1: How do we avoid generalizing to something that isn't true?

1. Look at the proof.

```
¬∃ x:ℚ, x * x = 2 :=

irrat_def' 2 fun h =>
  Exists.casesOn h fun a h =>
    Exists.casesOn h fun b h =>
      And.casesOn h fun copr h =>
        let_fun a_div := let_fun c := (Prime.dvd_mul Int.prime_two).mp
              (dvd_iff_exists_eq_mul_right.mpr
                (Exists.intro (b * b)
                  (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc 2 b b).symm))
                    (Eq.mpr (id (congrArg (fun _a => _a = 2 * b * b) h)) (Eq.refl (2 * b * b))))));
          Or.casesOn (motive := fun t => c = t → 2 ∣ a) c (fun h h_1 => h) (fun h h_1 => h) (Eq.refl c);
        let_fun a_is_pk := dvd_iff_exists_eq_mul_right.mp a_div;
        Exists.casesOn a_is_pk fun k hk =>
          let_fun h := (Eq.mp (congrArg (fun _a => _a * _a = 2 * b * b) hk) h).symm;
          let_fun b_div := let_fun c :=(Prime.dvd_mul Int.prime_two).mp
                (dvd_iff_exists_eq_mul_left.mpr
                  (Exists.intro (k * k)
                    (Eq.mp (congrArg (fun _a => b * b = _a) (mul_assoc k k 2).symm)
                      (Eq.mp (congrArg (fun _a => b * b = k * _a) (mul_comm 2 k))
                        ((Int.mul_eq_mul_left_iff (Prime.ne_zero Int.prime_two)).mp
                          (Eq.mp (congrArg (fun _a => 2 * (b * b) = _a) (mul_assoc 2 k (2 * k)))
                            (Eq.mp (congrArg (fun _a => _a = 2 * k * (2 * k)) (mul_assoc 2 b b)) h)))))));
            Or.casesOn (motive := fun t => c = t → 2 ∣ b) c (fun h h_1 => h) (fun h h_1 => h) (Eq.refl c);
          let_fun p_dvd_gcd := (dvd_gcd_iff 2 a b).mpr { left := a_div, right := b_div };
          Prime.not_dvd_one Int.prime_two (Eq.mp (congrArg (fun _a => 2 ∣ _a) copr) p_dvd_gcd)
```

Problem #1: How do we avoid generalizing to something that isn't true?

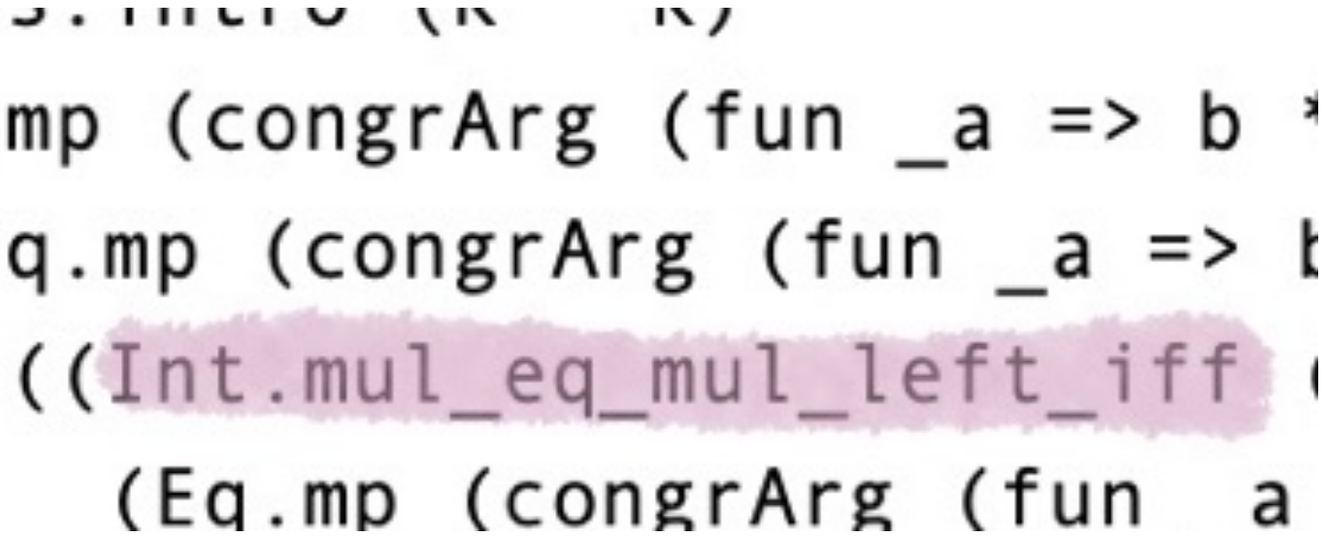
1. Look at the proof term. It contains many identifiers (lemmas or inference rules).

```
¬∃ x:ℚ, x * x = 2 :=

irrat_def' 2 fun h =>
  Exists.casesOn h fun a h =>
    Exists.casesOn h fun b h =>
      And.casesOn h fun copr h =>
        let_fun a_div := let_fun c := (Prime.dvd_mul Int.prime_two).mp
              (dvd_iff_exists_eq_mul_right.mpr
                (Exists.intro (b * b)
                  (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc 2 b b).symm))
                    (Eq.mpr (id (congrArg (fun _a => _a = 2 * b * b) h)) (Eq.refl (2 * b * b))))));
          Or.casesOn (motive := fun t => c = t → 2 ∣ a) c (fun h h_1 => h) (fun h h_1 => h) (Eq.refl c);
        let_fun a_is_pk := dvd_iff_exists_eq_mul_right.mp a_div;
        Exists.casesOn a_is_pk fun k hk =>
          let_fun h := (Eq.mp (congrArg (fun _a => _a * _a = 2 * b * b) hk) h).symm;
          let_fun b_div := let_fun c :=(Prime.dvd_mul Int.prime_two).mp
                (dvd_iff_exists_eq_mul_left.mpr
                  (Exists.intro (k * k)
                    (Eq.mp (congrArg (fun _a => b * b = _a) (mul_assoc k k 2).symm)
                      (Eq.mp (congrArg (fun _a => b * b = k * _a) (mul_comm 2 k))
                        ((Int.mul_eq_mul_left_iff (Prime.ne_zero Int.prime_two)).mp
                          (Eq.mp (congrArg (fun _a => 2 * (b * b) = _a) (mul_assoc 2 k (2 * k)))
                            (Eq.mp (congrArg (fun _a => _a = 2 * k * (2 * k)) (mul_assoc 2 b b)) h)))))));
            Or.casesOn (motive := fun t => c = t → 2 ∣ b) c (fun h h_1 => h) (fun h h_1 => h) (Eq.refl c);
          let_fun p_dvd_gcd := (dvd_gcd_iff 2 a b).mpr { left := a_div, right := b_div };
          Prime.not_dvd_one Int.prime_two (Eq.mp (congrArg (fun _a => 2 ∣ _a) copr) p_dvd_gcd)
```

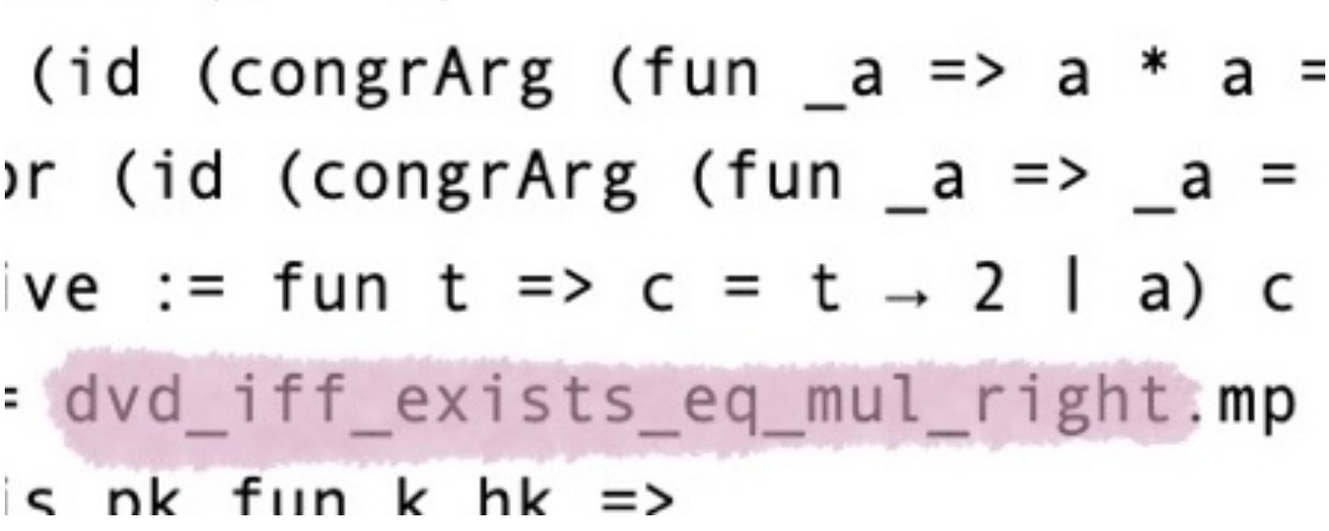
Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or an inference rule) in the proof…



Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or an inference rule) in the proof…



Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or an inference rule) in the proof…

```
p (congrArg (fun _a => 2 *
.mp (congrArg (fun _a => _a
:= fun t => c = t → 2 ∣ b)
(dvd_gcd_iff 2 a b).mpr {
```

Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or an inference rule) in the proof, examine its statement (type).

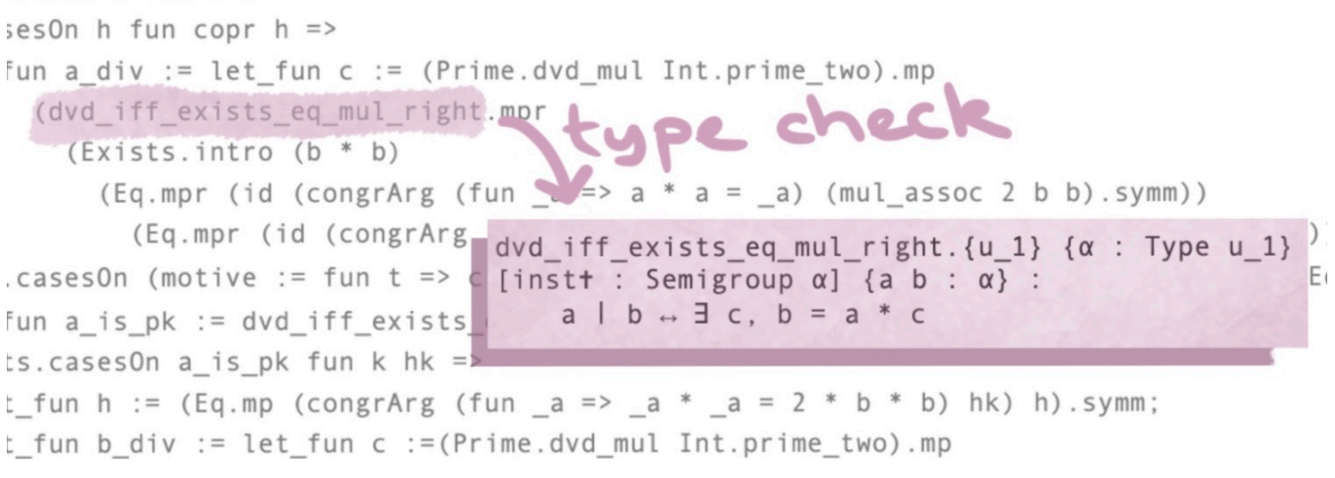
```
Prime.dvd_mul.{u_1} {α : Type u_1} [inst✝ : CommMonoidWithZero α]
{p : α} (hp : Prime p) {a b : α} :
    p ∣ a * b ↔ p ∣ a ∨ p ∣ b
```

type check

```
: := (Prime.dvd_mul Int.prime_two).mp
ıul_right.mpr
* b)
```

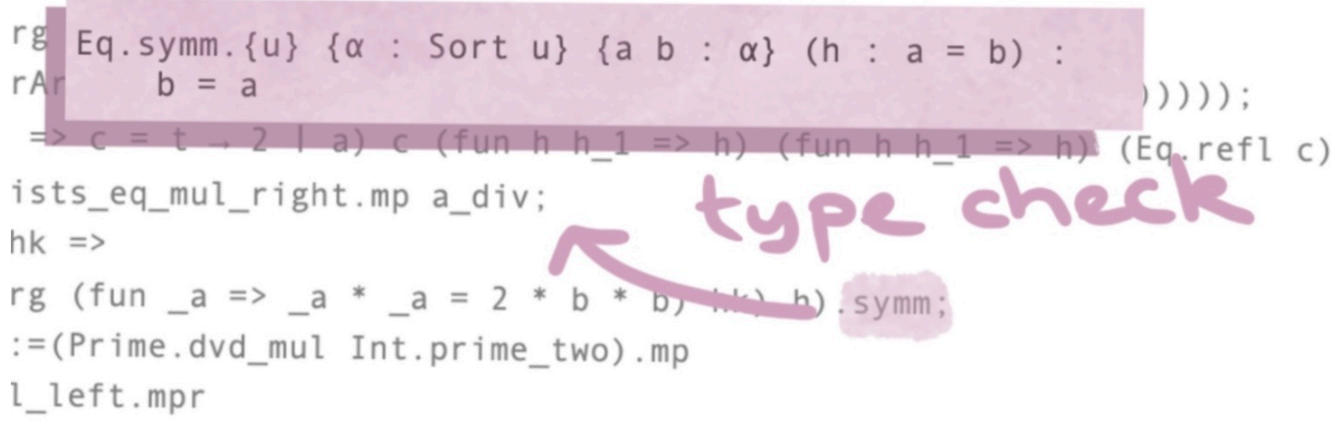
Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or an inference rule) in the proof, examine its statement.



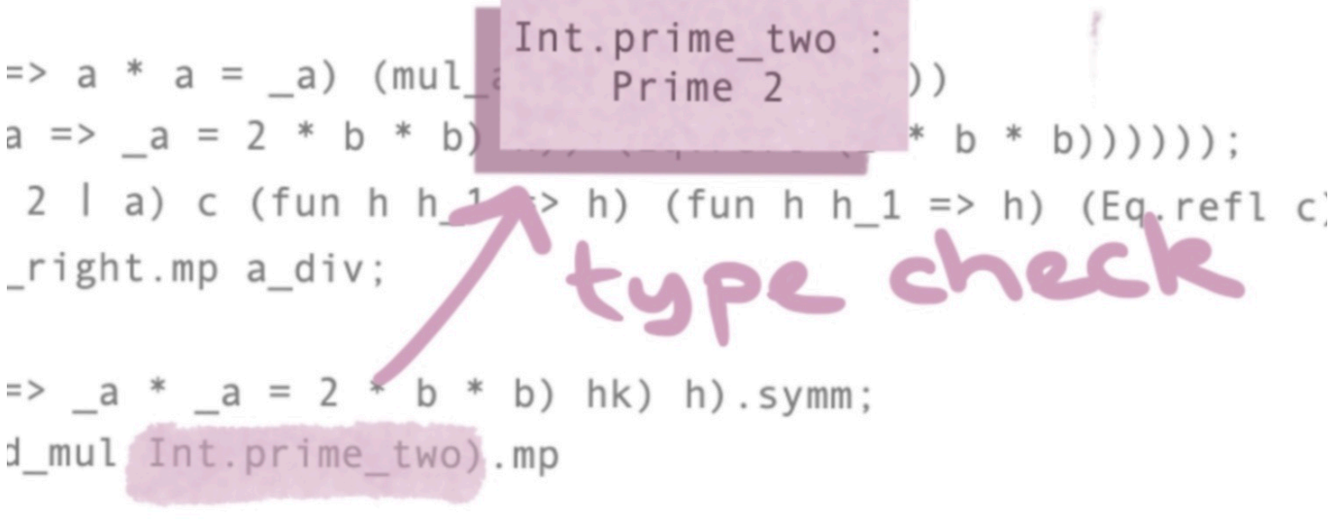
Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or inference rule) in the proof, examine its statement.



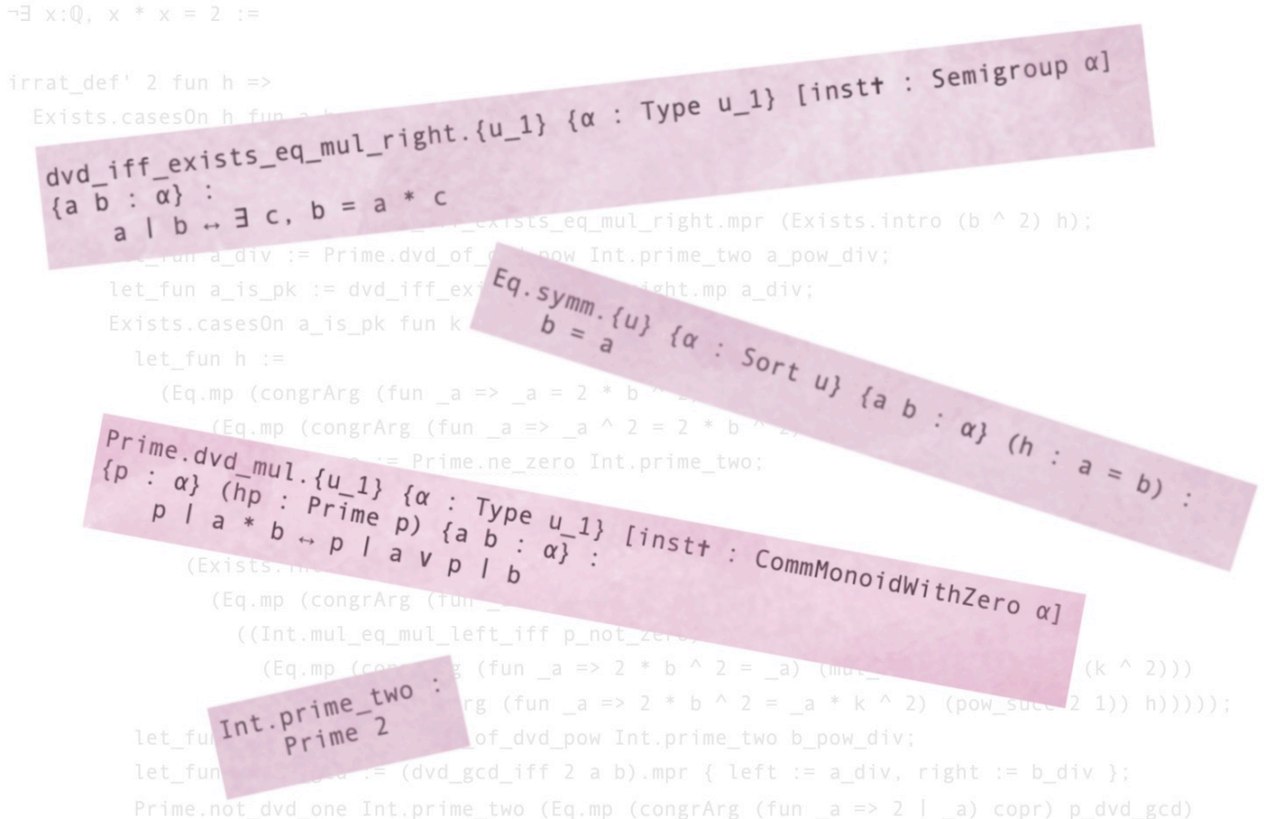
Problem #1: How do we avoid generalizing to something that isn't true?

1. For each identifier (a lemma or inference rule) in the proof, examine its statement.



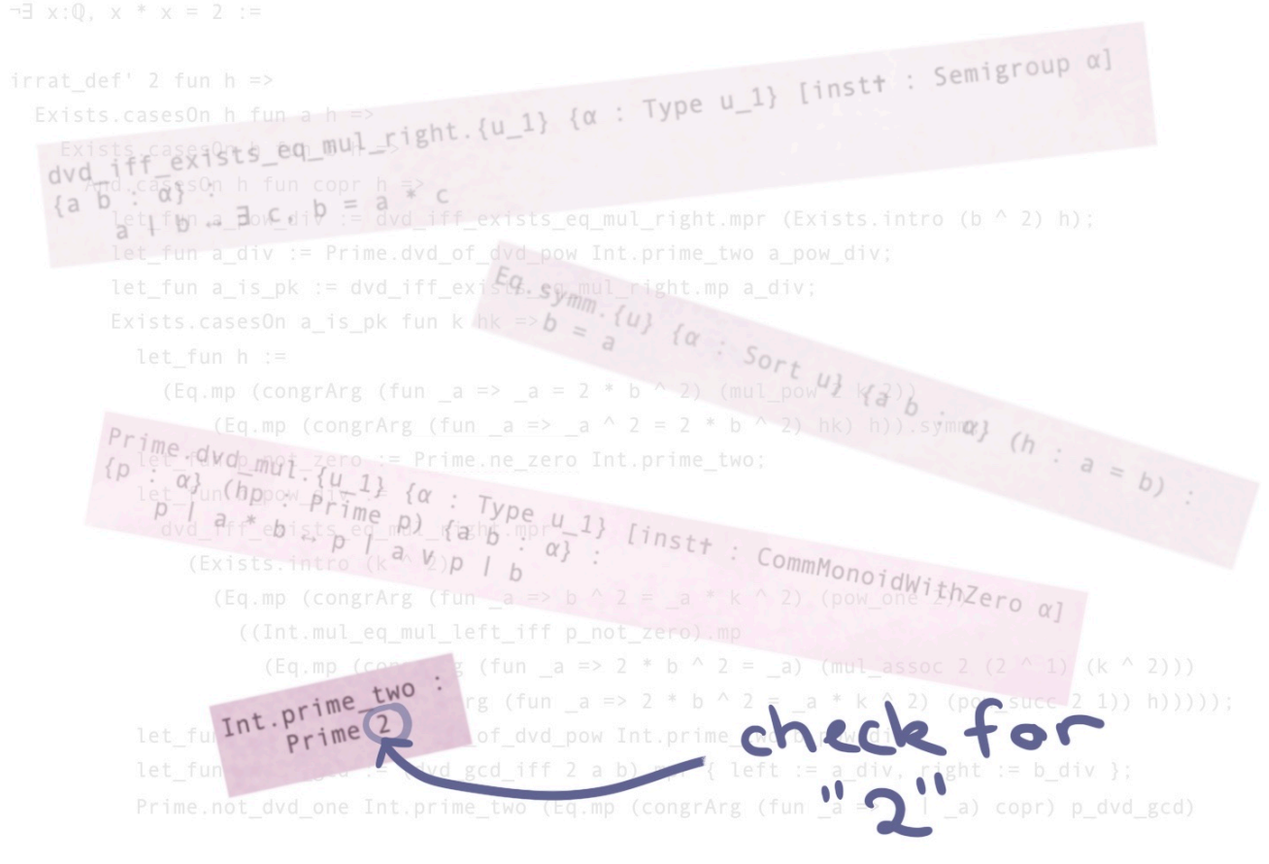
Problem #1: How do we avoid generalizing to something that isn't true?

2. Collect all identifiers whose statement contains the expression e to be generalized.



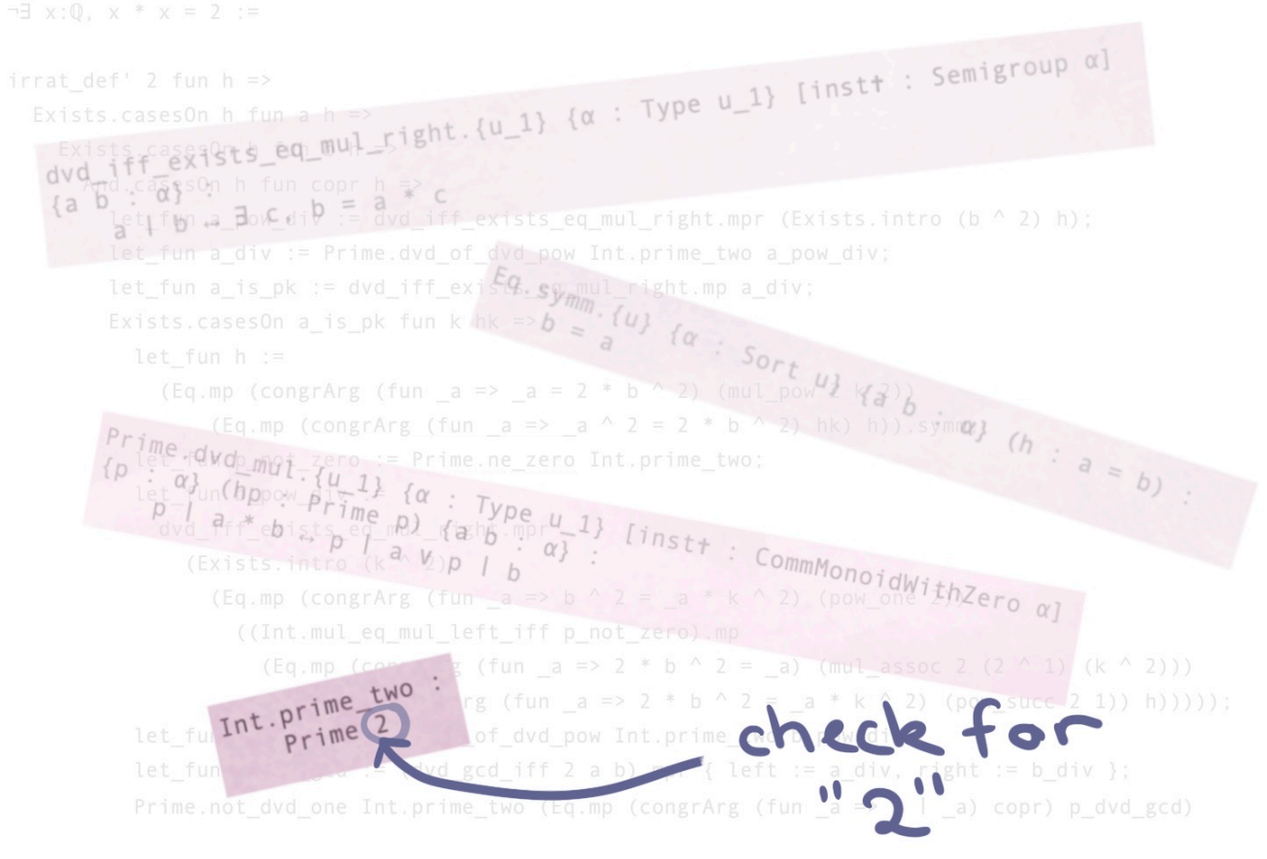
Problem #1: How do we avoid generalizing to something that isn't true?

2. Collect all identifiers whose statement contains the expression e to be generalized.



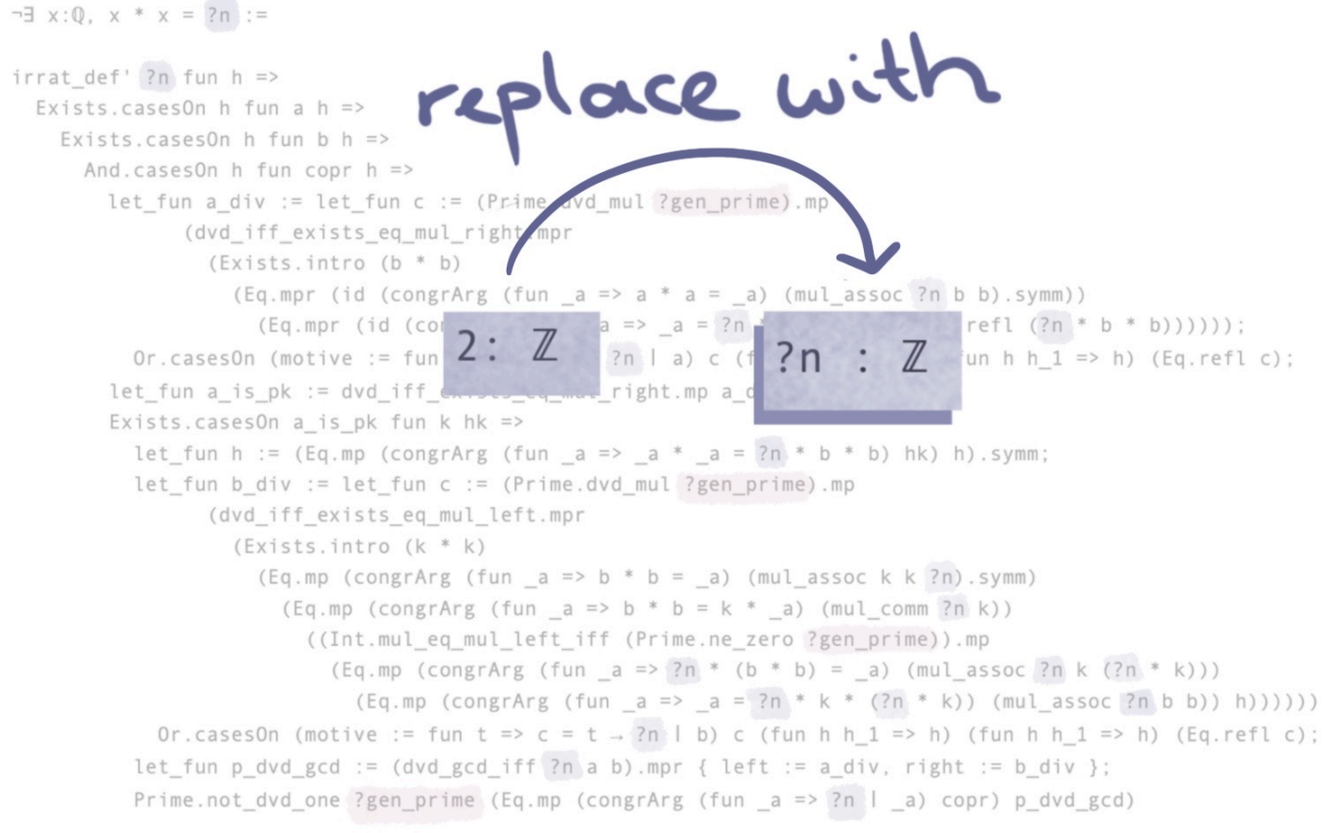
Problem #1: How do we avoid generalizing to something that isn't true?

2. Collect all identifiers whose statement (type) contains the expression e to be generalized.



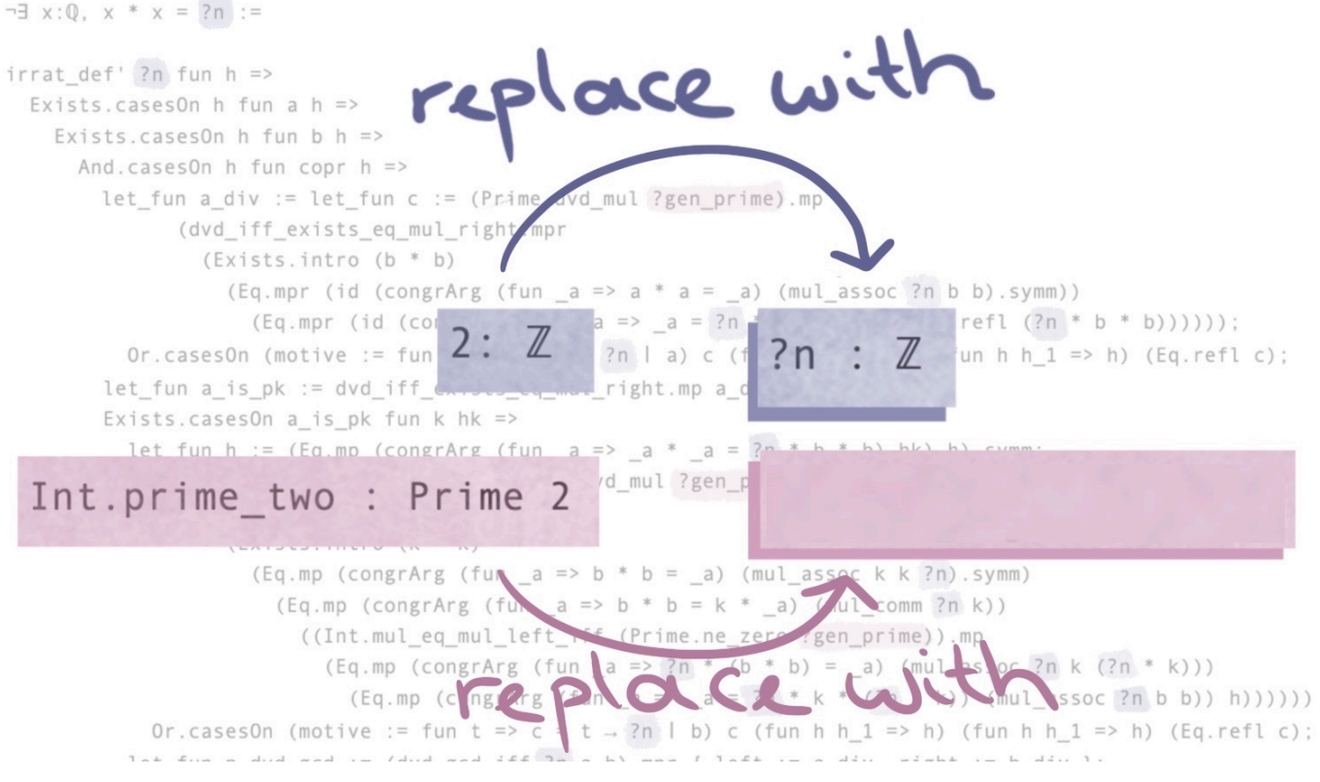
Problem #1: How do we avoid generalizing to something that isn't true?

3. Replace all instances of e with a metavariable (hole) with the same type.



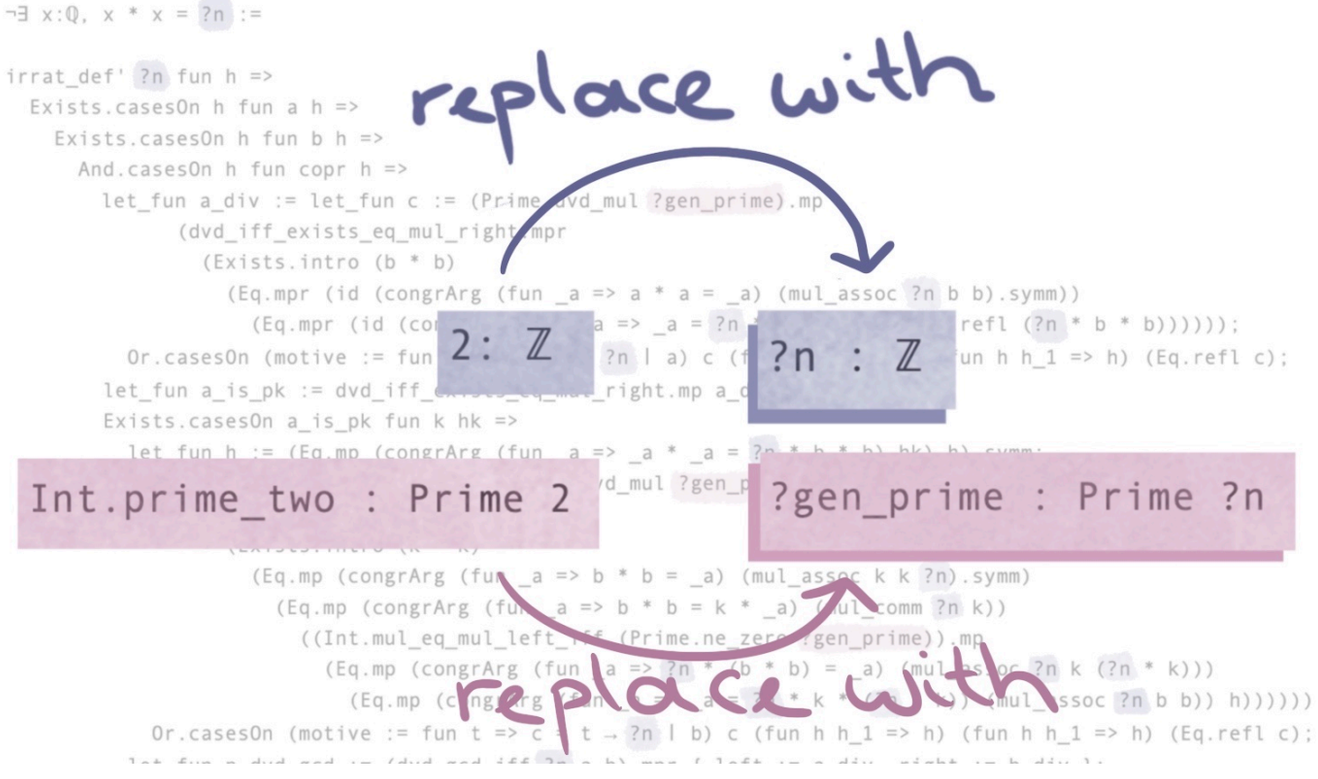
Problem #1: How do we avoid generalizing to something that isn't true?

4. For any identifier whose statement contains e (that is, for any proposition about e used in the proof)…



Problem #1: How do we avoid generalizing to something that isn't true?

4. For any identifier whose statement contains e (that is, for any proposition about e used in the proof), replace it with a metavariable representing the generalized proposition.



Problem #1: How do we avoid generalizing to something that isn't true?

5. Add the metavariables (holes) as local hypotheses.

```
¬∃ x:ℚ, x * x = ?n :=

irrat_def' ?n fun h =>
    . . .
    congrArg (fun _a => _a * _a = ?n * b * b) hk) h)
    . . .
    (Prime.dvd_mul ?gen_prime).mp
    . . .
```

- Remember

```
?n : ℤ

?gen_prime : Prime ?n
```

Problem #1: How do we avoid generalizing to something that isn't true?

5. Add the metavariables (holes) as local hypotheses.

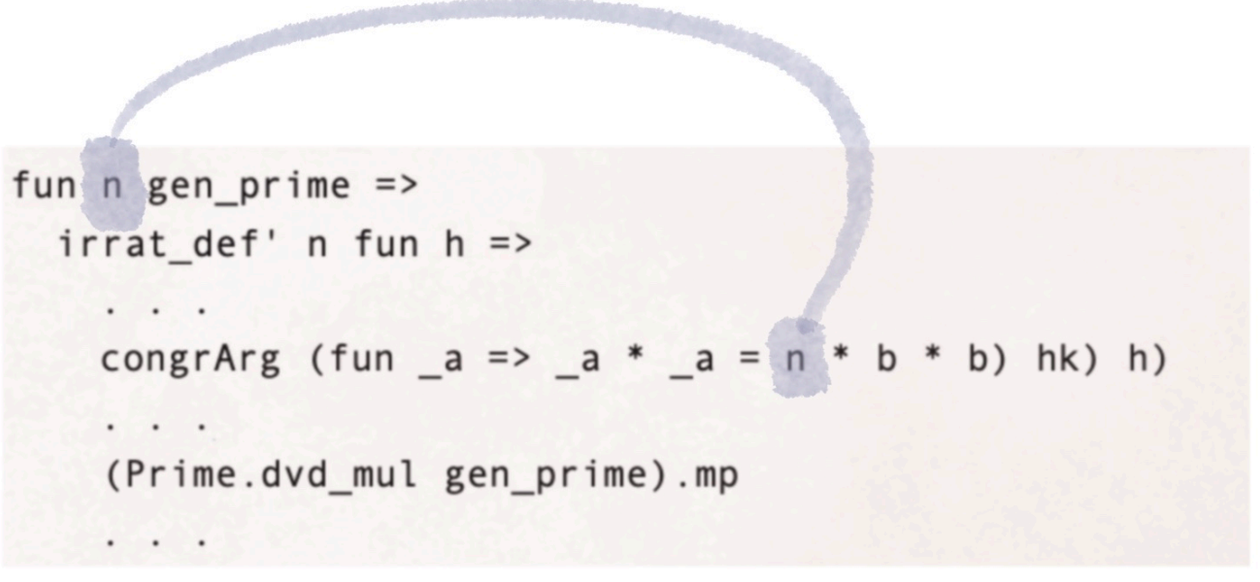
```
irrat_def' ?n fun h =>
    . . .
    congrArg (fun _a => _a * _a = ?n * b * b) hk) h)
    . . .
    (Prime.dvd_mul ?gen_prime).mp
    . . .
```

- Remember

```
?n : ℤ
?gen_prime : Prime ?n
```

Problem #1: How do we avoid generalizing to something that isn't true?

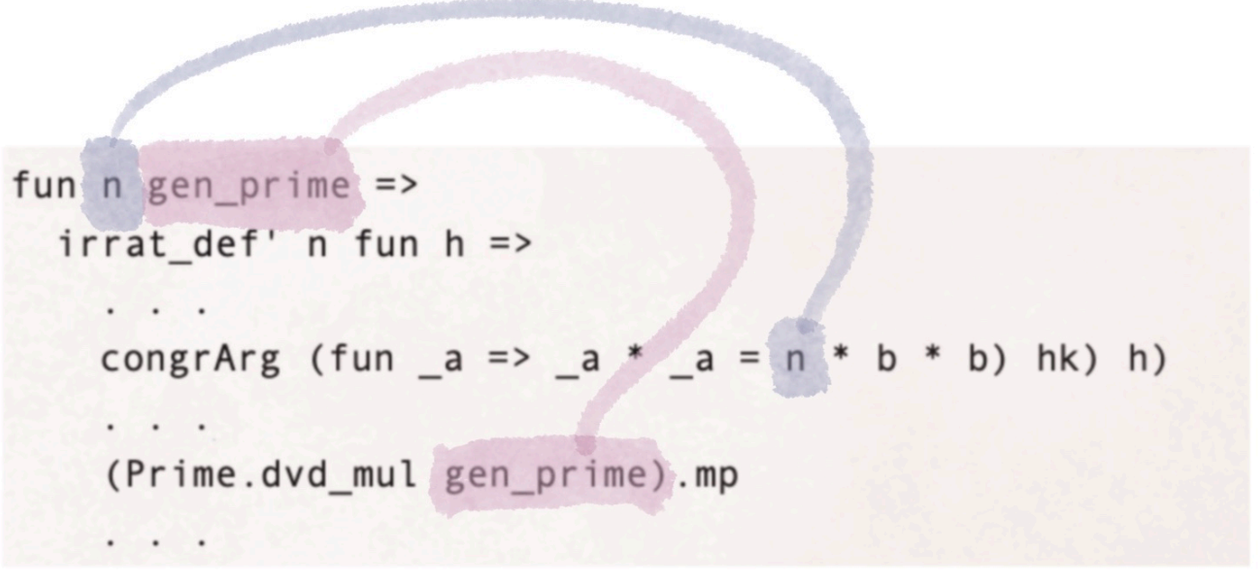
5. Add the metavariables (holes) as local hypotheses.



```
fun n gen_prime =>
  irrat_def' n fun h =>
    . . .
    congrArg (fun _a => _a * _a = n * b * b) hk) h)
    . . .
    (Prime.dvd_mul gen_prime).mp
    . . .
```

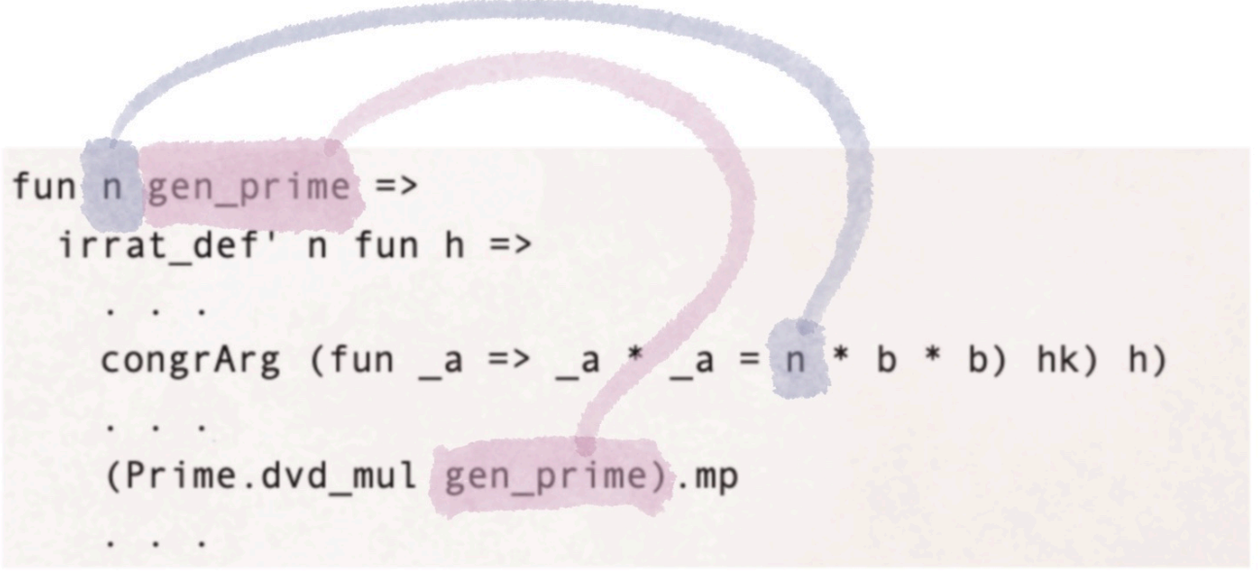
Problem #1: How do we avoid generalizing to something that isn't true?

5. Add the metavariables (holes) as local hypotheses.



Problem #1: How do we avoid generalizing to something that isn't true?

5. Add the metavariables (holes) as local hypotheses. (That is, we abstract the metavariables in the proof term into bound variables.)



Problem #1: How do we avoid generalizing to something that isn't true?

6. Determine the statement of the generalized theorem (In type theory, this can be done automatically by inferring the type of the generalized proof.)

```
∀ (n : ℤ), Prime n → ¬∃ x:ℚ, x * x = n :=

fun n gen_prime =>
  irrat_def' n fun h =>
    . . .
    congrArg (fun _a => _a * _a = n * b * b) hk) h)
    . . .
    (Prime.dvd_mul gen_prime).mp
    . . .
```

Summary: Given …

- a proof `p` of theorem `t`
- an expression `e` to generalize in `t`

…the generalization algorithm works as follows:

1. Look at the proof term `p`. For each identifier in the proof, examine its statement.
2. Collect all identifiers whose statement contains the expression `e` to be generalized.
3. Replace all instances of `e` in the proof term with a metavariable of the same type.
4. For any identifier whose statement contains `e`, replace it with a metavariable representing the generalized proposition.
5. Add the metavariables (holes) as local hypotheses.
6. Determine the statement of the generalized theorem.

Problem #1: How do we avoid generalizing to something that isn't true?

The idea here is very simple. Given …

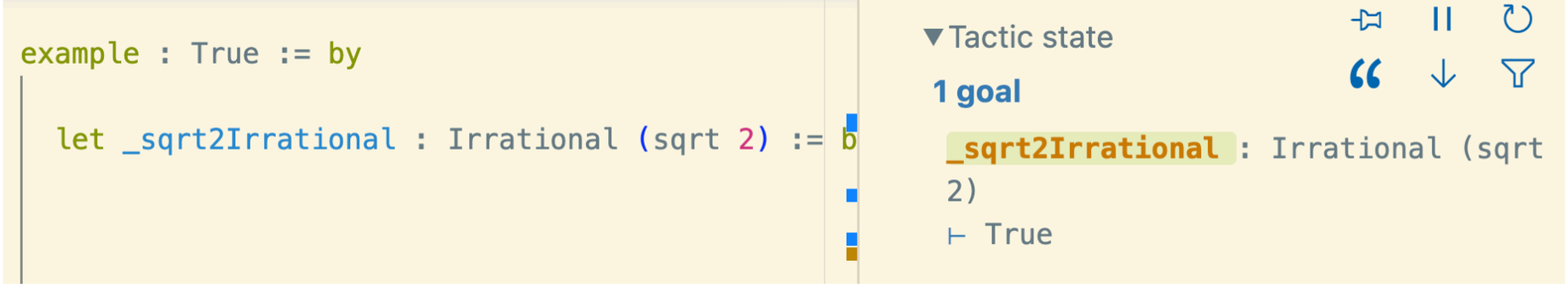
- a proof
- an expression e to generalize in the proof

…the generalization algorithm works as follows:

- if we ever use a fact about e in the proof, then we add any facts we used about e as hypotheses of the generalized proof. (This lets us reuse most of the original proof.)

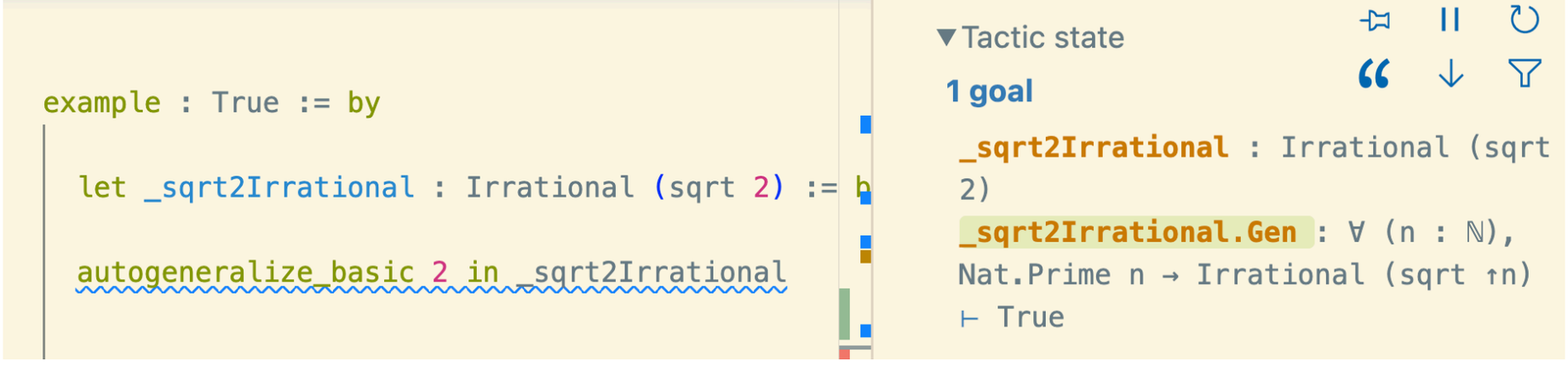
But…sometimes this generalization isn't general enough.

This generalization mechanism works fine for this example…



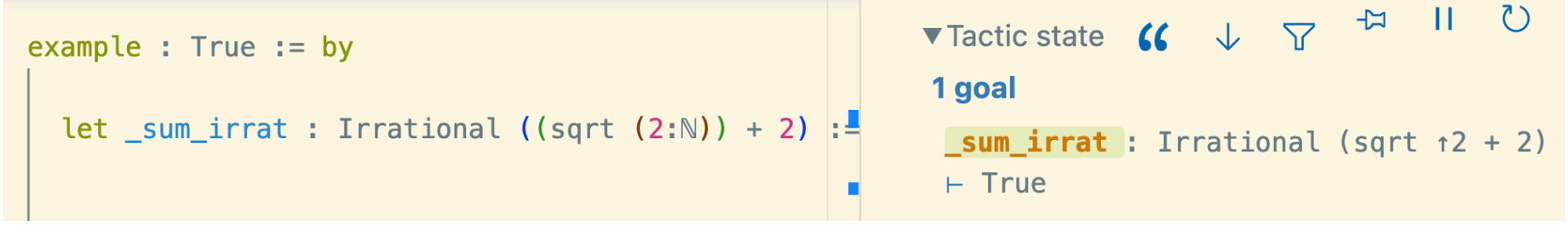
But…sometimes this generalization isn't general enough.

This generalization mechanism works fine for this example…



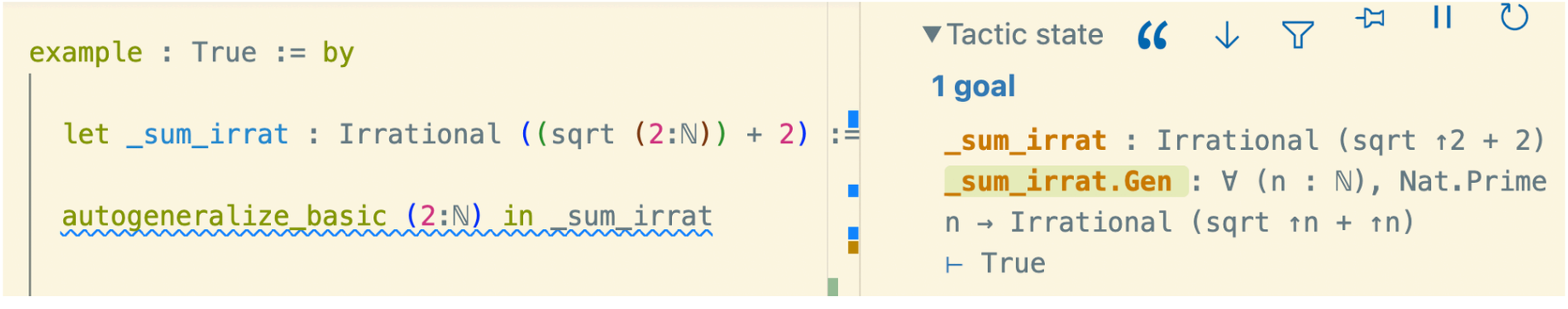
But…sometimes this generalization isn't general enough.

But when the expression to generalize (e.g. "2") occurs multiple times in the statement…



But…sometimes this generalization isn't general enough.

But when the expression to generalize (e.g. "2") occurs multiple times in the statement…this mechanism generalizes each occurrence to the same variable.

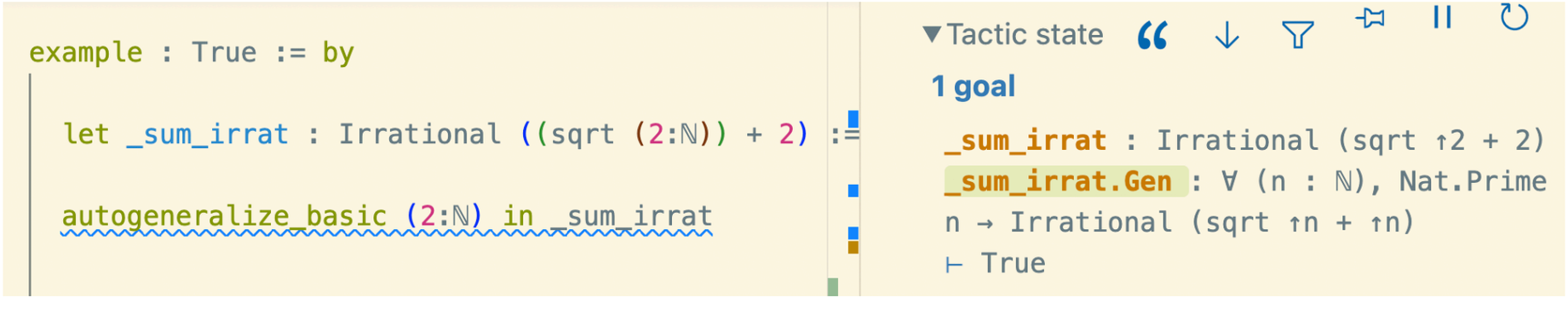


This is saying:

$$\text{For all primes } p,\ \sqrt{p} + p \text{ is irrational.}$$

But…sometimes this generalization isn't general enough.

But when the expression to generalize (e.g. "2") occurs multiple times in the statement…this mechanism generalizes each occurrence to the same variable.



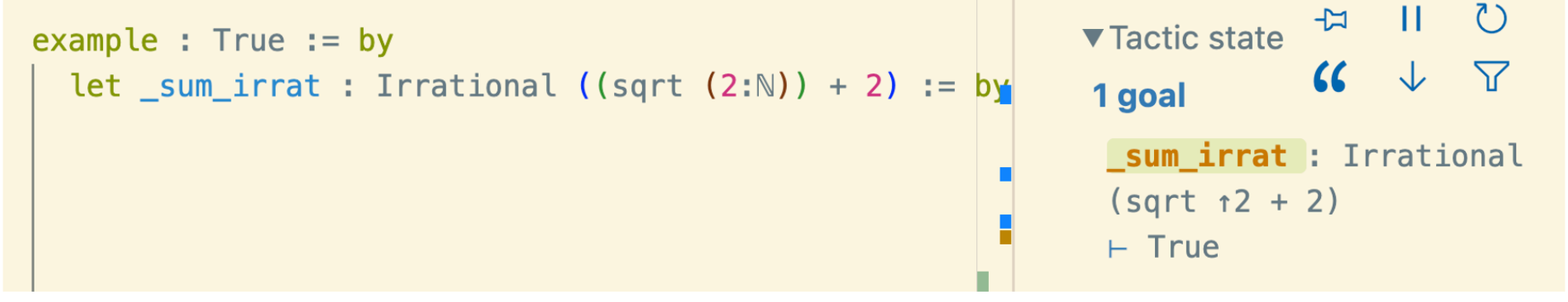
This is saying:

$$\text{For all primes } p, \sqrt{p} + p \text{ is irrational.}$$

The above result is technically true, but *doesn't take full advantage of the proof* to determine a more general statement.

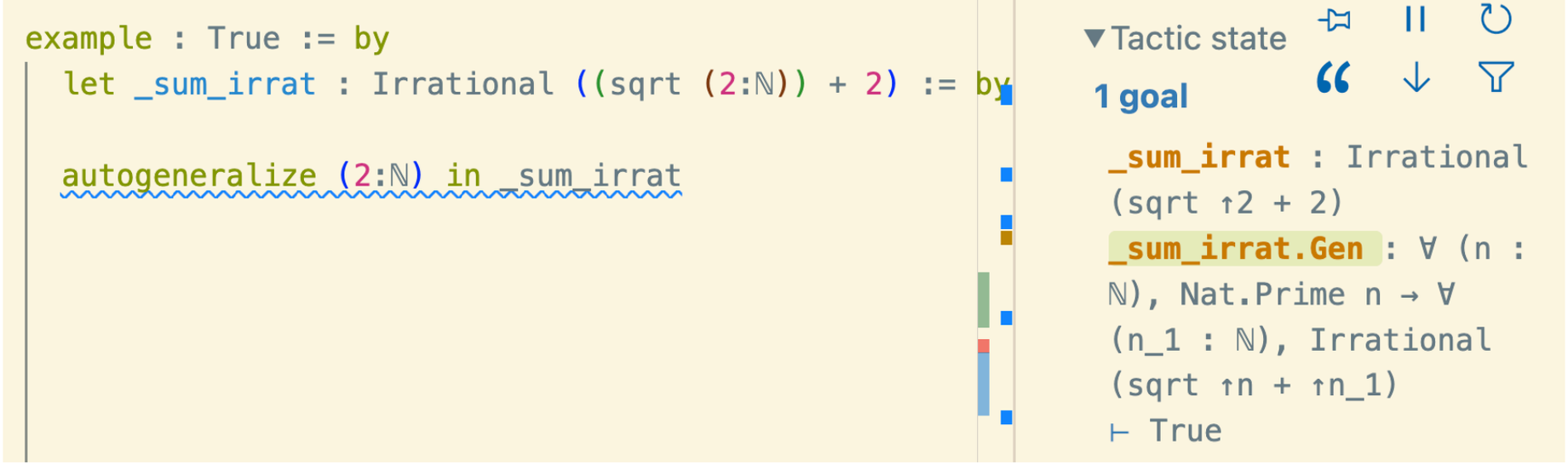
But…sometimes this generalization isn't general enough.

So we build on Pons's mechanism (which we've implemented in Lean as `autogeneralize_basic`)…



But…sometimes this generalization isn't general enough.

So we build on Pons's mechanism (which we've implemented in Lean as `autogeneralize_basic`)…And create a new mechanism to generalize theorems more robustly (and name it `autogeneralize`).

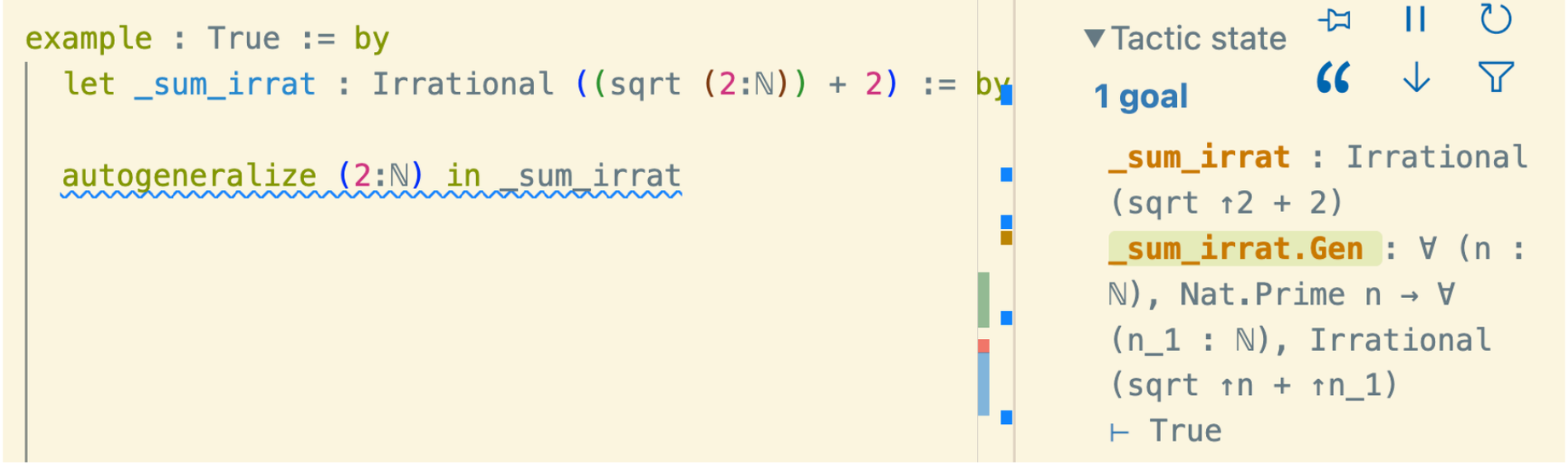


This is saying:

$$\text{For any prime } p \text{ and natural number } n, \sqrt{p} + n \text{ is irrational.}$$

But…sometimes this generalization isn't general enough.

So we build on Pons's mechanism (which we've implemented in Lean as `autogeneralize_basic`)…And create a new mechanism to generalize theorems more robustly (and name it `autogeneralize`).



We go into details on how this more robust mechanism works in the following slides.

Problem #2: How do we avoid generalizing to something overly specific?

We need to disambiguate constants, somehow.

Problem #2: How do we avoid generalizing to something overly specific?

We need to disambiguate constants, somehow.

Intuitively, to do this, we “check which occurrences of the constant are linked by the proof”.

Problem #2: How do we avoid generalizing to something overly specific?

We need to disambiguate constants, somehow.

More technically, but still somewhat vaguely, we “check which metavariables unify in the generalized proof.”

Problem #2: How do we avoid generalizing to something overly specific?

We need to disambiguate constants, somehow.

More technically, but still somewhat vaguely, we “check which metavariables unify in the generalized proof.”

Let’s go into details in an example.

Problem #2: How do we avoid generalizing to something overly specific?

First, given a proof term…

```
Irrational.add_nat 2
  (irrat_def 2 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul Nat.prime_two).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc 2 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = 2 * b * b) h)) (Eq.refl (2 * b * b))))));
                  ...
```

Problem #2: How do we avoid generalizing to something overly specific?

First, given a proof term, generalize each instance of e to *different* metavariables.

```
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

Problem #2: How do we avoid generalizing to something overly specific?

First, given a proof term, generalize each instance of e to *different* metavariables.

```
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

Intuitively, this means every time we see a “2” in the proof, we abstract it to *different* arbitrary number.

Problem #2: How do we avoid generalizing to something overly specific?

First, given a proof term, generalize each instance of e to *different* metavariables.

```
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

Intuitively, this means every time we see a “2” in the proof, we abstract it to *different* arbitrary number. But of course, this gives us a proof that barely makes sense.

Problem #2: How do we avoid generalizing to something overly specific?

First, given a proof term, generalize each instance of e to *different* metavariables.

```
∀ (x_0 : ℕ), Prime(x_0),∀ (x_1: ℕ), ∀ (x_2: ℕ),∀ (x_3: ℕ) ... ∀ (x_484 : ℕ),
    Irrational (sqrt x_0 + x_484) :=
...
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

One way you can see how nonsensical this is: when we add all of these holes as local hypotheses…we get way too many local hypothesis (more than 484!).

Problem #2: How do we avoid generalizing to something overly specific?

How to fix it?

```
∀ (x_0 : ℕ), Prime(x_0),∀ (x_1: ℕ), ∀ (x_2: ℕ),∀ (x_3: ℕ) ... ∀ (x_484 : ℕ),
    Irrational (sqrt x_0 + x_484) :=
...
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

Problem #2: How do we avoid generalizing to something overly specific?

Intuitively: we want to make sure whenever we prove something about the variable under the square root, we use the same variable. And whenever we prove something about the variable outside the square root, we use a consistently different variable.

```
∀ (x_0 : ℕ), Prime(x_0),∀ (x_1: ℕ), ∀ (x_2: ℕ),∀ (x_3: ℕ) ... ∀ (x_484 : ℕ),
    Irrational (sqrt x_0 + x_484) :=
...
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

Problem #2: How do we avoid generalizing to something overly specific?

Concretely – we need to **unify metavariables** in the proof.

How this can be done in a type-theoretic framework (the hard way):

- Recursively visit every function application $f(a_1, \ldots, a_n)$ in the proof term.
- We know
  - the types of the arguments of f
  - the inferred types of the arguments $a_1, \ldots, a_n$
- Compare the the types of f's arguments with the type of the a_is, and set matching metavariables equal to each other.

Problem #2: How do we avoid generalizing to something overly specific?

Concretely —we need to **unify metavariables** in the proof.

How this can be done in a type-theoretic framework (the easy way):

- In Lean (and possibly other type-theoretic languages), simply running a type check on the proof term will unify metavariables that need to be linked.

Problem #2: How do we avoid generalizing to something overly specific?

Concretely – we need to **unify metavariables** in the proof.

```
Irrational.add_nat ?m.1
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.0 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.0 * b * b) h)) (Eq.refl (?m.0 * b * b))))));
                    ...
```

Problem #2: How do we avoid generalizing to something overly specific?

Concretely – we need to **unify metavariables** in the proof. After that, we can abstract the metavariables as before to get a more general theorem.

```
∀ (x_0 : ℕ), Nat.Prime x_0 → ∀ (x_1: ℕ),
    Irrational (sqrt x_0 + x_1) :=
...
Irrational.add_nat ?m.1
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.0 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.0 * b * b) h)) (Eq.refl (?m.0 * b * b))))));
                    ...
```

Problem #2: How do we avoid generalizing to something overly specific?

And now we have the result shown previously.

```
example : True := by
  let _sum_irrat : Irrational ((sqrt (2:ℕ)) + 2) := by

  autogeneralize (2:ℕ) in _sum_irrat
```

▼Tactic state

1 goal

```
_sum_irrat : Irrational
(sqrt ↑2 + 2)
_sum_irrat.Gen : ∀ (n :
ℕ), Nat.Prime n → ∀
(n_1 : ℕ), Irrational
(sqrt ↑n + ↑n_1)
⊢ True
```

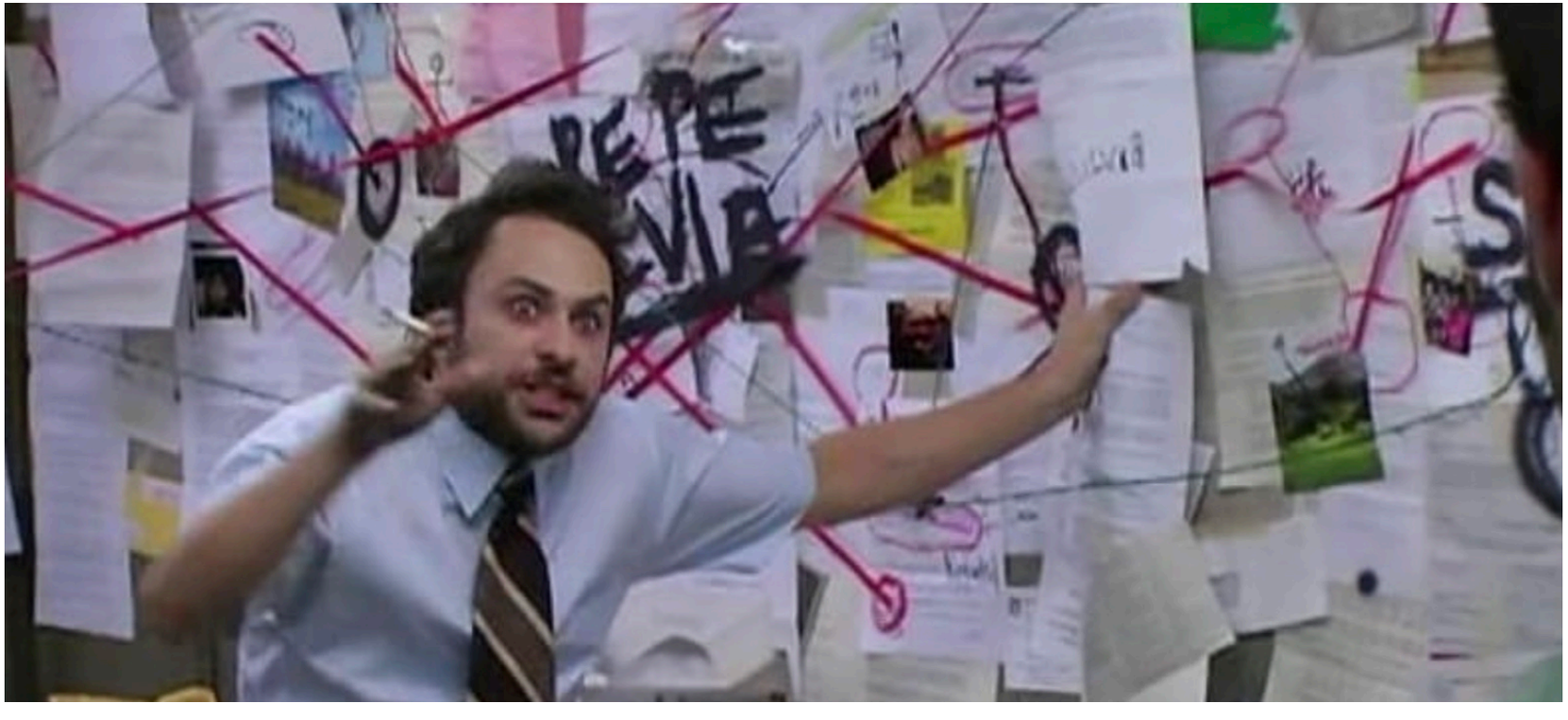
Problem #2: How do we avoid generalizing to something overly specific?

In summary, we make the type checker play the role of the conspiracy theorist…

```
∀ (x_0 : ℕ), Prime(x_0),∀ (x_1: ℕ), ∀ (x_2: ℕ),∀ (x_3: ℕ)…∀ (x_484 : ℕ),
    Irrational (sqrt x_0 + x_484) :=
...
Irrational.add_nat ?m.484
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.1 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.2 * b * b) h)) (Eq.refl (?m.3 * b * b))))))));
                  ...
```

Problem #2: How do we avoid generalizing to something overly specific?

In summary, we make the type checker play the role of the conspiracy theorist…



Problem #2: How do we avoid generalizing to something overly specific?

In summary, we make the type checker play the role of the conspiracy theorist…

```
∀ (x_0 : ℕ), Nat.Prime x_0 → ∀ (x_1: ℕ),
    Irrational (sqrt x_0 + x_1) :=
...
Irrational.add_nat ?m.1
  (irrat_def ?m.0 fun h =>
    Exists.casesOn h fun a h =>
      Exists.casesOn h fun b h =>
        And.casesOn h fun copr h => let_fun a_div := let_fun c :=
          (Nat.Prime.dvd_mul ?gen_prime).mp (dvd_iff_exists_eq_mul_right.mpr
            (Exists.intro (b * b)
              (Eq.mpr (id (congrArg (fun _a => a * a = _a) (mul_assoc ?m.0 b b).symm))
                (Eq.mpr (id (congrArg (fun _a => _a = ?m.0 * b * b) h)) (Eq.refl (?m.0 * b * b))))));
                    ...
```

Problem #2: How do we avoid generalizing to something overly specific?

The overview of how this more robust mechanism works is as follows.

Problem #2: How do we avoid generalizing to something overly specific?

Given …

- a proof `p` of theorem `t`
- an expression `e` to generalize in `t`

Problem #2: How do we avoid generalizing to something overly specific?

The `autogeneralize` tactic…

1. For each identifier in the proof `p`, examines its statement.
2. Collects all identifier statements containing the `e` to be generalized.
3. Replaces all instances of `e` in the proof term with **different metavariables** of the same type.
4. **Unifies linked metavariables (by recursively unifying at function applications in the proof term.)**
5. For any identifier whose statement contains `e`, replaces it with a metavariable representing the generalized proposition.
6. Adds the metavariables (holes) as local hypotheses.
7. Determines the statement of the generalized theorem.

Problem #2: How do we avoid generalizing to something overly specific?

We can see algorithm in action in the example we mentioned at the beginning, with counting the number of functions from a set of size 3 to a set of size 3.

Problem #2: How do we avoid generalizing to something overly specific?

We can see algorithm in action in the example we mentioned at the beginning, with counting the number of functions from a set of size 3 to a set of size 3.

```
example :=
by
  let fun_set : Fintype.card α = 3 → Fintype.card β = 3 →
```

```
fun_set : Fintype.card α = 3 →
Fintype.card β = 3 →
Fintype.card (α → β) = 3 ^ 3
```

Problem #2: How do we avoid generalizing to something overly specific?

We can see this working in the example we mentioned at the beginning, with counting the number of functions from a set of size 3 to a set of size 3.

Here is the way **the basic algorithm** (`autogeneralize_basic`) designed by Pons would generalize it.

```
example :=
by
  let fun_set : Fintype.card α = 3 → Fintype.card β = 3 →

  autogeneralize_basic 3 in fun_set
```

```
fun_set : Fintype.card α = 3 →
Fintype.card β = 3 →
Fintype.card (α → β) = 3 ^ 3
fun_set.Gen : ∀ (n : ℕ),
Fintype.card α = n →
Fintype.card β = n →
Fintype.card (α → β) = n ^ n
```

Problem #2: How do we avoid generalizing to something overly specific?

We can see this working in the example we mentioned at the beginning, with counting the number of functions from a set of size 3 to a set of size 3.

And here's how our **more robust algorithm** (`autogeneralize`) would generalize it.

```
example :=
by
  let fun_set : Fintype.card α = 3 → Fintype.card β = 3 →

  autogeneralize 3 in fun_set
```

```
fun_set : Fintype.card α = 3 →
Fintype.card β = 3 →
Fintype.card (α → β) = 3 ^ 3
fun_set.Gen : ∀ (n n_1 : ℕ),
Fintype.card α = n →
Fintype.card β = n_1 →
Fintype.card (α → β) = n_1 ^ n
```

Problem #2: How do we avoid generalizing to something overly specific?

We can see this working in the example we mentioned at the beginning, with counting the number of functions from a set of size 3 to a set of size 3.

And here's how our **more robust algorithm** (`autogeneralize`) would generalize it.

```
example :=
by
  let fun_set : Fintype.card α = 3 → Fintype.card β = 3 →

  autogeneralize 3 in fun_set
```

```
fun_set : Fintype.card α = 3 →
Fintype.card β = 3 →
Fintype.card (α → β) = 3 ^ 3
fun_set.Gen : ∀ (n n_1 : ℕ),
Fintype.card α = n →
Fintype.card β = n_1 →
Fintype.card (α → β) = n_1 ^ n
```

Takeaway: ***When tactics detect relevant instances of a manipulated variable*** *through unification, they* ***mimic a more human-oriented style of mathematics.***

A tradeoff between elegance and generality

Consider this theorem (originally in the Pons paper).

```
mult_permute : ∀ (n m p : ℕ),
n * (m * p) = m * (n * p)
```

A tradeoff between elegance and generality

Consider this theorem (originally in the Pons paper).

```
mult_permute : ∀ (n m p : ℕ),
n * (m * p) = m * (n * p)
```

But this theorem holds for more binary operators than just multiplication.

A tradeoff between elegance and generality

```
mult_permute : ∀ (n m p : ℕ),
n * (m * p) = m * (n * p)
```

If we generalize all occurrences of multiplication to the same constant, we get a nice generalization to the fact that this equality holds for any binary operator that is commutative and associative.

```
example := by
  let mult_permute :  ∀ (n m p : ℕ), n * (m * p)

  autogeneralize_basic Mul.mul in mult_permute
```

```
mult_permute : ∀ (n m p : ℕ), n * (m
* p) = m * (n * p)
mult_permute.Gen : ∀ (f : ℕ → ℕ → ℕ),
  (∀ (n m k : ℕ), f (f n m) k = f n
(f m k)) → (∀ (n m : ℕ), f n m = f m
n) → ∀ (n m p : ℕ), f n (f m p) = f m
(f n p)
```

A tradeoff between elegance and generality

```
mult_permute : ∀ (n m p : ℕ),
n * (m * p) = m * (n * p)
```

If we generalize each of the four occurrences of multiplication separately, we get a generalization to **four different binary operators**, and **conditions about joint associativity and commutativity specific to each operator**.

```
example := by
  let mult_permute :  ∀ (n m p : ℕ), n * (m * p)

  autogeneralize Mul.mul in mult_permute
```

```
mult_permute : ∀ (n m p : ℕ), n * (m
* p) = m * (n * p)
mult_permute.Gen : ∀ (f f_1 f_2 f_3 :
ℕ → ℕ → ℕ),
  (∀ (n m k : ℕ), f_2 (f_3 n m) k = f
n (f_1 m k)) →
    (∀ (n m : ℕ), f_3 n m = f_3 m n)
→ ∀ (n m p : ℕ), f n (f_1 m p) = f m
(f_1 n p)
```

This abstraction is more general than Pons's. But probably less elegant.

A tradeoff between elegance and generality

But of course, the same generalization behavior that seems undesirable in this context (putting unnecessary hypotheses on *every* occurrence of the generalized $*$)…

A tradeoff between elegance and generality

But of course, the same generalization behavior that seems undesirable in this context (putting unnecessary hypotheses on *every* occurrence of the generalized $*$)…

- …is desirable in the $Irrational(\sqrt{2}+2)$ context (putting unnecessary hypotheses on *every* occurrence of the generalized 2).

A tradeoff between elegance and generality

But of course, the same generalization behavior that seems undesirable in this context (putting unnecessary hypotheses on *every* occurrence of the generalized $*$)…

- …is desirable in the $Irrational(\sqrt{2}+2)$ context (putting unnecessary hypotheses on *every* occurrence of the generalized 2).
- …and may desirable in the context of vector spaces, where joint associativity between two operators e.g. a vector-vector operator and a scalar-vector operator.

Ideally, we want this **choice of generalization to be informed by the context** , to strike the right **balance between elegance and generality**.

Computation rules vs. Deduction rules

We can prove that 2 times 3 is even by using **deduction rules** saying that 2 times any number is even.

```
example := by
  let two_times_three_is_even : Even (2*3) := by
    unfold Even; apply Exists.intro 3; rw [two_mul]
```

1 goal

two_times_three_is_even : Even (2 * 3)

Computation rules vs. Deduction rules

We can prove that 2 times 3 is even by using **deduction rules** saying that 2 times any number is even.

```
example := by
  let two_times_three_is_even : Even (2*3) := by
    unfold Even; apply Exists.intro 3; rw [two_mul]
```

```
1 goal
two_times_three_is_even : Even (2 * 3)
```

And we can generalize the “3” in this proof.

```
example := by
  let two_times_three_is_even : Even (2*3) := by
    unfold Even; apply Exists.intro 3; rw [two_mul]

  autogeneralize 3 in two_times_three_is_even
```

```
1 goal
two_times_three_is_even : Even (2 * 3)
two_times_three_is_even.Gen : ∀ (n : ℕ), Even (2 * n)
```

Computation rules vs. Deduction rules

But we can also prove that 2 times 3 is even by **computing** that 2 times 3 equals 6, and then saying that 6 is even.

```
example := by
  let two_times_three_is_even : Even (2*3) := by
    simp only [Nat.reduceMul]; apply six_is_even
```

1 goal

two_times_three_is_even : Even (2 * 3)

Computation rules vs. Deduction rules

But we can also prove that 2 times 3 is even by **computing** that 2 times 3 equals 6, and then saying that 6 is even.

```
example := by
  let two_times_three_is_even : Even (2*3) := by
    simp only [Nat.reduceMul]; apply six_is_even
```

```
1 goal
two_times_three_is_even : Even
(2 * 3)
```

And we can *not* generalize the "3" in this proof.

```
example := by
  let two_times_three_is_even : Even (2*3) := by
    simp only [Nat.reduceMul]; apply six_is_even

  autogeneralize 3 in two_times_three_is_even
```

```
AutoGeneralizeDemo3000.lean:86:2
The type of the
proof doesn't match the
statement.  Perhaps a
computation rule was used?
```

Computation rules vs. Deduction rules

Why?

If some part of the proof is done by **computation** or evaluation, then in Lean, the steps used in that part of proof **does not show in the proof term**.

Computation rules vs. Deduction rules

Why?

If some part of the proof is done by **computation** or evaluation, then in Lean, the steps used in that part of proof **does not show in the proof term**.

This is a downside of a language like Lean (which doesn't show these computations in the proof term).

- But, this general issue (i.e. that some theorem-proving tactics are black boxes) is addressed and tackled in other languages like Beluga (by Brigitte Pientka).
- So, theorems proved in these languages may be especially amenable to proof-based generalization.

Comparison to existing generalizers

We find it helpful to divide generalization algorithms into two categories:

- Risky generalizers
- Proof-based generalizers

Comparison to existing generalizers

We find it helpful to divide generalization algorithms into two categories:

- Risky generalizers…
  - …generalize to statements that **may or may not be true**.
    As such, they often rely on conjecture-disprovers to dismiss incorrect conjectures.
- Proof-based generalizers
  - …only generalize to **true statements**.

Comparison to existing generalizers

We find it helpful to divide generalization algorithms into two categories:

- Risky generalizers…
  - …generalize to statements that **may or may not be true**. As such, they often rely on conjecture-disprovers to dismiss incorrect conjectures
  - …for example
    - Boyer and Moore's Generalizer in "A Computational Logic" (1979).
    - Ireland and Bundy's Generalization Proof Critic in "Productive Use of Failure" (1996).
- Proof-based generalizers
  - …only generalize to **true statements**.

Comparison to existing generalizers

We find it helpful to divide generalization algorithms into two categories:

- Risky generalizers…
  - …generalize to statements that **may or may not be true**.
    As such, they often rely on conjecture-disprovers to dismiss incorrect conjectures
- Proof-based generalizers
  - …only generalize to **true statements**.
  - for example
    - Best's Typeclass Generalizer in "Automatically Generalizing Theorems Using Typeclasses" (2021).
    - Pons's Generalizer in "Generalization in type theory based proof assistants" (2000).
    - Our `autogeneralize` tactic.

When should we use proof-based generalization?

When should we use proof-based generalization?

The most obvious use case is *proof reuse*.

When should we use proof-based generalization?

The most obvious use case is *proof reuse*.

```
Irrational (sqrt ↑2 + 2)
```

When should we use proof-based generalization?

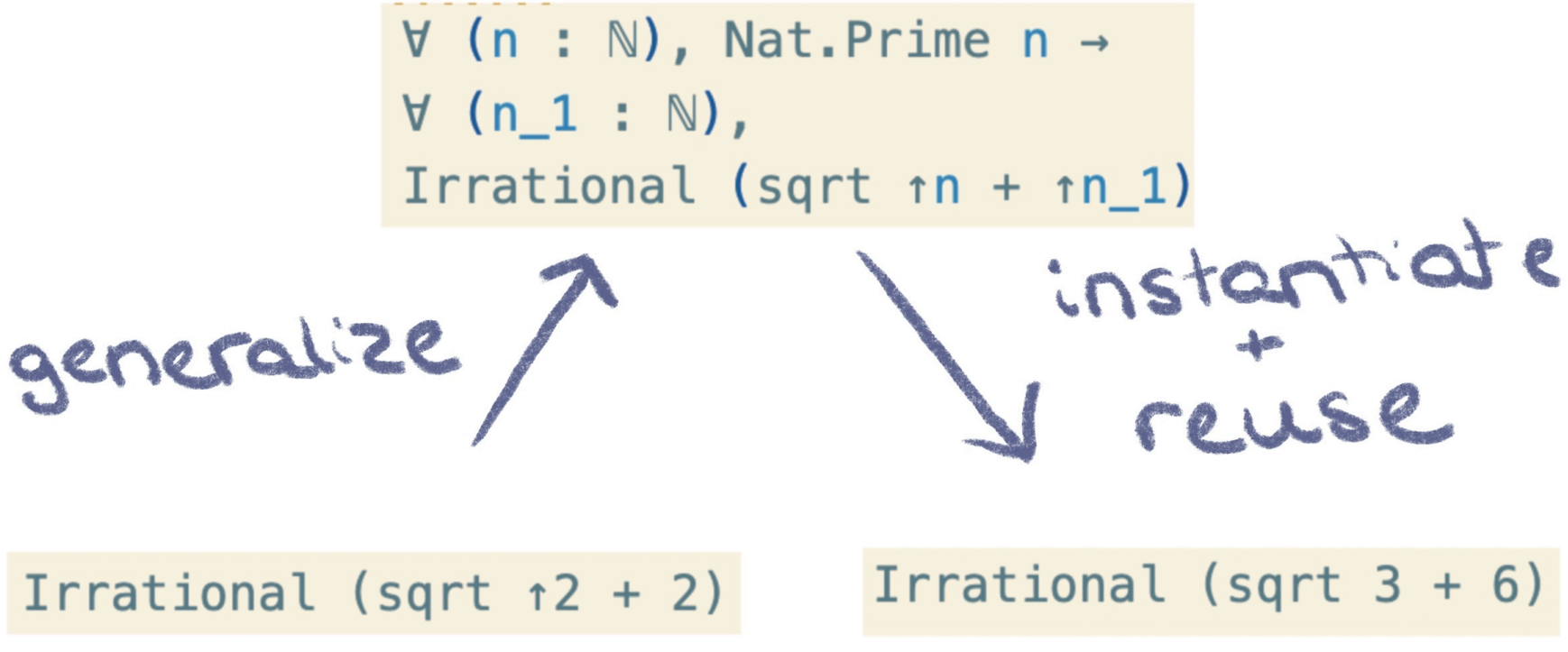
The most obvious use case is *proof reuse*.

```
∀ (n : ℕ), Nat.Prime n →
∀ (n_1 : ℕ),
Irrational (sqrt ↑n + ↑n_1)
```


```
Irrational (sqrt ↑2 + 2)
```

When should we use proof-based generalization?

The most obvious use case is *proof reuse*.



When should we use proof-based generalization?

In action:

```
example : Irrational (sqrt 3 + 6) := by
  let sum_irrat : Irrational (sqrt (2:ℕ) + 2) :=
  autogeneralize (2:ℕ) in sum_irrat
```

```
sum_irrat : Irrational (sqrt ↑2 + 2)
sum_irrat.Gen : ∀ (n : ℕ), Nat.Prime
n → ∀ (n_1 : ℕ), Irrational (sqrt ↑n
+ ↑n_1)
⊢ Irrational (sqrt 3 + 6)
```

When should we use proof-based generalization?

In action:

```
example : Irrational (sqrt 3 + 6) := by
  let sum_irrat : Irrational (sqrt (2:ℕ) + 2) :=
  autogeneralize (2:ℕ) in sum_irrat
  specialize sum_irrat.Gen 3 (Nat.prime_three) 6
```

```
sum_irrat : Irrational (sqrt ↑2 + 2)
sum_irrat.Gen : Irrational (sqrt ↑3
+ ↑6)
⊢ Irrational (sqrt 3 + 6)
```

When should we use proof-based generalization?

In action:

```
example : Irrational (sqrt 3 + 6) := by
  let sum_irrat : Irrational (sqrt (2:ℕ) + 2) :=
  autogeneralize (2:ℕ) in sum_irrat
  specialize sum_irrat.Gen 3 (Nat.prime_three) 6
  assumption
```

No goals

▶ All Messages (93)

When should we use proof-based generalization?

But in developing this tactic, we had a very specific use case in mind…

When should we use proof-based generalization?

But in developing this tactic, we had a very specific use case in mind…
learning from failure.

When should we use proof-based generalization?

By “failure” to prove a conjecture C, we mean:

- **Not** that the “failure” is an *invalid proof* of C, and the “learning” is a *patch*.
- **Rather**, that the “failure” is a *valid proof* of the negation $\neg C$ (e.g. a counterexample to C), and the “learning” is a *generalization* of that proof that helps decrease the size of the proof search space.

Here’s an example…

When should we use proof-based generalization?

Suppose we want to answer: *Which polynomials with real coefficients have a real root?*

We might notice that **all degree-1 polynomials have a real root**, and generate the following conjecture…

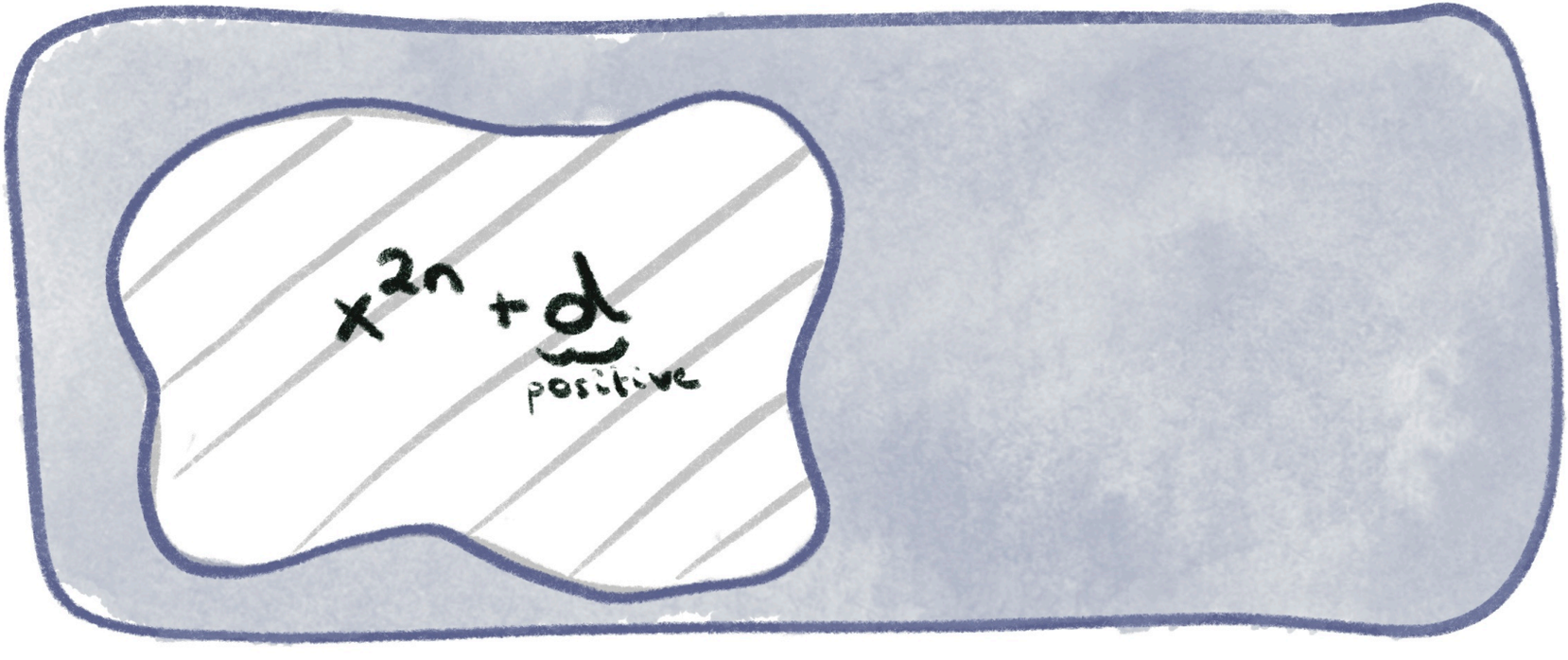
When should we use proof-based generalization?

Which polynomials with real coefficients have a real root? We might…

- **Generate a conjecture:** Every polynomial with real coefficients has a real root.
- **Fail:** The polynomial $x^2 + 1$ has no real root.
- **Generalize (learn from) the failure :** Any polynomial of the form $x^n + d$, where n is even and $d > 0$ has no real root , by the `autogeneralized` proof.
- **Apply what you learned:** It's not straightforward to see whether an even-degree polynomial has a real root. We may want to turn our attention to polynomials of odd-degree.

Polynomials with Real Coefficients and Real Roots

Polynomials with Real Coefficients and Real Roots

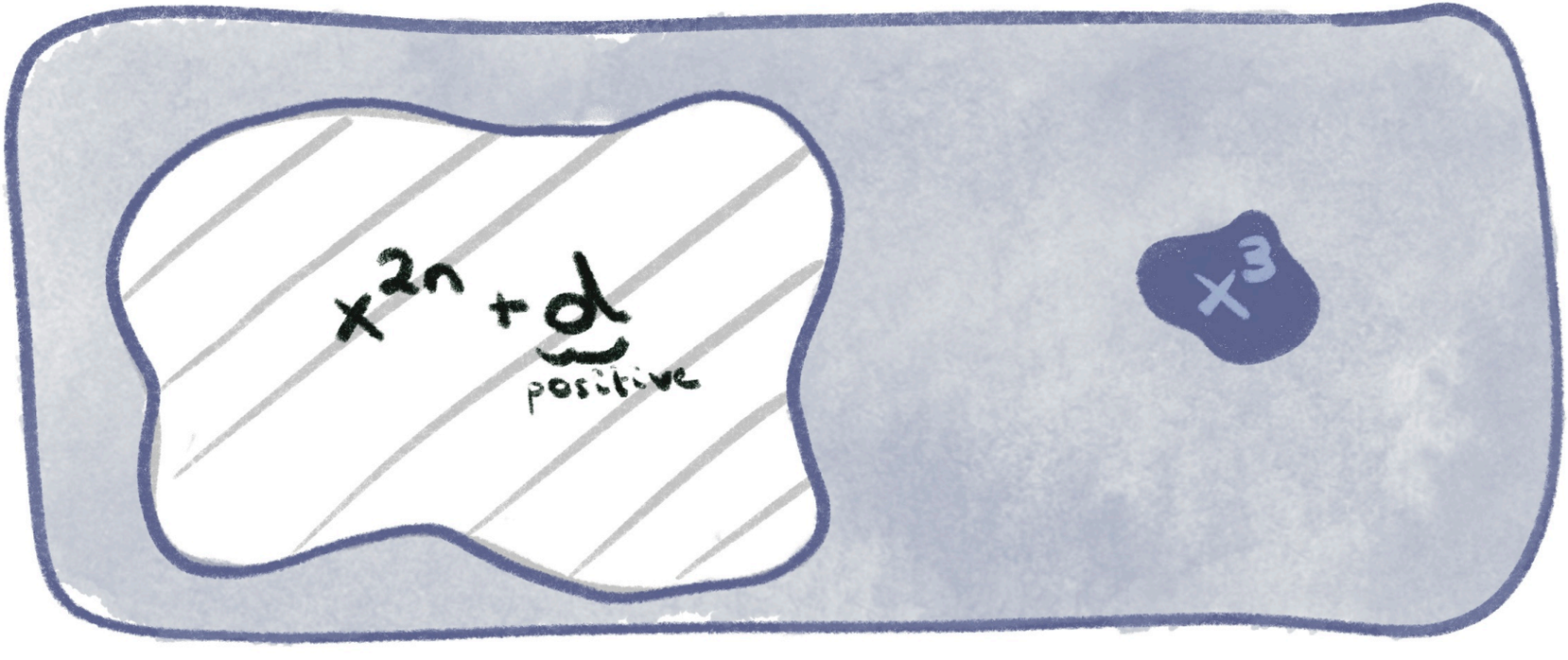


When should we use proof-based generalization?

We found even-degree polynomials with no real root… is it possible that we could find a degree 3 polynomial with no real root?

- **Generate a conjecture:** There exists a cubic with no real root.
- **Fail:** Every cubic polynomial has a real root. (Proof: The cubic term dominates when $x \ll -1$ and dominates when $1 \ll x$, and switches sign between these two cases. So, by intermediate value theorem, the polynomial must have a real root).
- **Generalize (learn from) the failure :** Any odd-degree polynomial has a real root, by the `autogeneralized` proof.
- **Apply what you learned:** We can now drastically cut down on our search space – we can safely restrict our attention to characterizing which even-degree polynomials have a real root.

Polynomials with Real Coefficients and Real Roots



Polynomials with Real Coefficients and Real Roots

When should we use proof-based generalization?

In general, the process is:

- Generate a conjecture.
- Fail.
- Generalize (learn from) the failure.
- Apply what you learned (for example, conjecturing the converse e.g. “no even-degree polynomial has a real root”)

When should we use proof-based generalization?

In general, the process is:

- Generate a conjecture.
- Fail.
- **Generalize (learn from) the failure**.
- Apply what you learned.

When should we use proof-based generalization?

In general, the process is:

- Generate a conjecture.
- Fail.
- **Generalize the failure** often means `autogeneralize`.
- Apply what you learned.

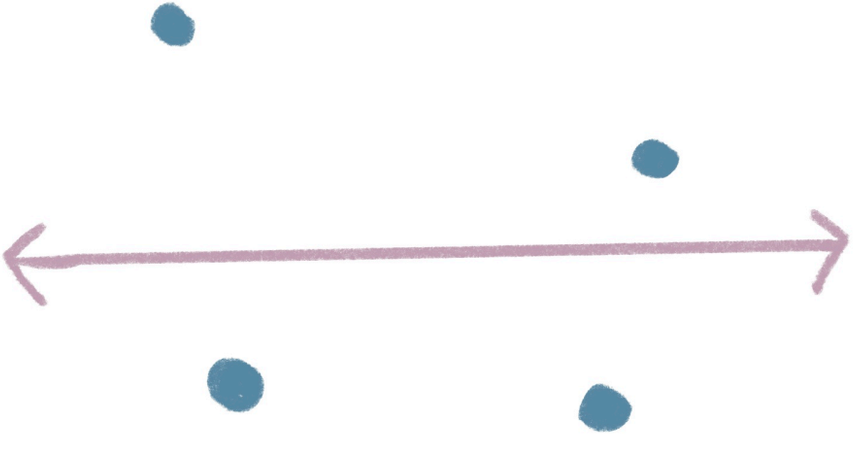
In math, **often "generalization" of failure is simply a "proof-based generalization" of failure**, as above. A mathematician will trace back through the proof, and notice the "same proof" holds for a much wider set of objects than the proof was applied for.

But maybe not everybody uses this exact proof-based generalization process to characterize polynomials with real roots…

Given $2n$ points on a plane, does there always exist a line such that n points are strictly on one side of the line, and n strictly on the other?

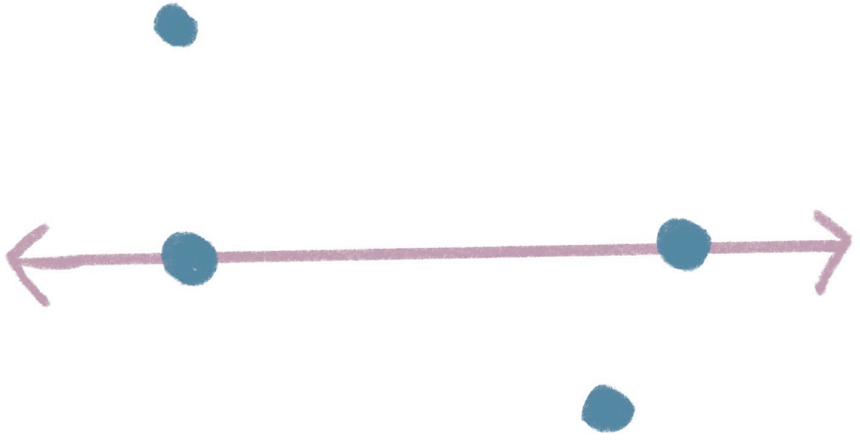
The typical problem-solving process is:

- **Generate a conjecture:** A moving line will pass through one point at a time. So, any appropriately translated line will bisect the set. Let's conjecture that there always exists a horizontal line which does.



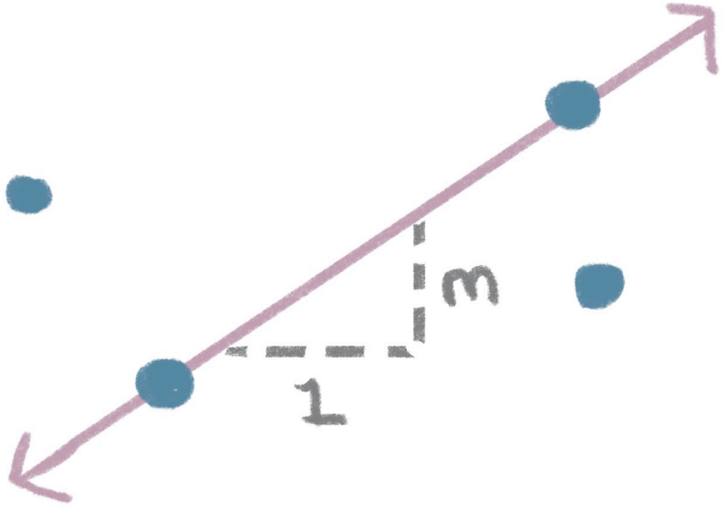
Given $2n$ points on a plane, does there always exist a line such that n points are strictly on one side of the line, and n strictly on the other?

- **Fail:** If the point set contains two points collinear through a horizontal line, the strategy fails (and there might not exist a horizontal line which bisects the set).



Given $2n$ points on a plane, does there always exist a line such that n points are strictly on one side of the line, and n strictly on the other?

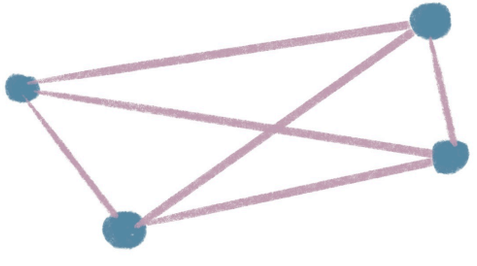
- **Generalize (learn from) the failure :** If the point set contains two points collinear through any line of gradient m, the strategy fails (i.e. there might not exist a line of gradient m which bisects the set).



Given $2n$ points on a plane, does there always exist a line such that n points are strictly on one side of the line, and n strictly on the other?

- **Apply what you learned:** Now, the strongest possible statement we could prove is the following strengthening of the inverse: If a point set contains *no* two points collinear through a line with gradient m, there *does* exist a line with gradient m to bisect it.

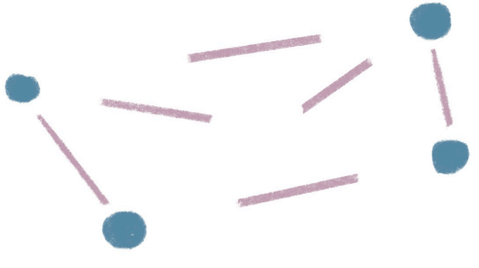
  Once proved, we can finish the proof by noticing at most $\binom{2n}{2}$ such "forbidden" gradients exist, and any line with a non-forbidden gradient will work.



Given $2n$ points on a plane, does there always exist a line such that n points are strictly on one side of the line, and n strictly on the other?

- **Apply what you learned:** Now, the strongest possible statement we could prove is the following strengthening of the inverse: If a point set contains *no* two points collinear through a line with gradient m, there *does* exist a line with gradient m to bisect it.

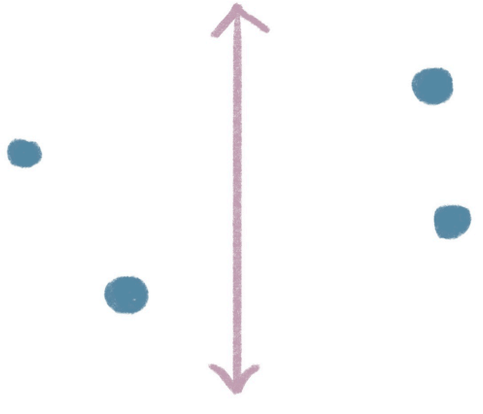
  Once proved, we can finish the proof by noticing at most $\binom{2n}{2}$ such "forbidden" gradients exist, and any line with a non-forbidden gradient will work.



Given $2n$ points on a plane, does there always exist a line such that n points are strictly on one side of the line, and n strictly on the other?

- **Apply what you learned:** Now, the strongest possible statement we could prove is the following strengthening of the inverse: If a point set contains *no* two points collinear through a line with gradient m, there *does* exist a line with gradient m to bisect it.

  Once proved, we can finish the proof by noticing at most $\binom{2n}{2}$ such "forbidden" gradients exist, and any line with a non-forbidden gradient will work.



The fact that so many people solve this problem in nearly the same way suggests there is **something quite natural about failure and proof-based generalization** of the failure.

When should we use proof-based generalization?

Proof-based generalization is also regularly used in research mathematics to refine conjectures.

Tim Gowers & Ryan Alweiss are currently working on an open problem – a particular strengthening of the cap-set conjecture. And they came up with the formulation of the problem using (instinctive) proof-based generalization.

The problem statement is:

- For any particular density $\delta \geq 0$, we can always find a dimension n big enough so that any subset of vector space F_3^n with at least density δ will contain a three-term arithmetic progression $\{x, x+d, x+2d\}$ such that the difference vector $d = e_i + e_j + e_k$ is a sum of 3 standard basis vectors.

How was this conjecture generated? Why the choice of 3 basis vectors?

When should we use proof-based generalization?

They started with the cap-set problem:

- For any particular density $\delta \geq 0$, we can always find a dimension n big enough so that any subset of F_3^n with density at least δ will contain a 3-term AP.

Then they did the following:

- **Generate a conjecture:** (Strengthening the above) It's possible to create such a three-term AP where the difference $d = e_i$ between points is given by **1** standard basis vector e_i.

When should we use proof-based generalization?

Then:

- **Fail:** For density $\delta = 1/3$, there is no vector space F_3^n where every $1/3$-dense subset contains such a 3-term AP. In particular, the $1/3$-dense subset made of the vectors whose components add to $0 \mod 3$ will contain no 3-term AP where the difference $d = e_i$ between points given by **1** standard basis vector.

When should we use proof-based generalization?

Then:

- **Generalize (learn from) the failure :** The same $1/3$-dense subset contains no 3-term AP when the difference d between points is given by $k \neq 0 \mod 3$ standard basis vectors.

When should we use proof-based generalization?

Then:

- **Apply what you learned:** They generated a new conjecture: A δ-dense subset contains a three-term AP where the difference $e_i + e_j + e_k$ between points is given by **3** standard basis vectors.

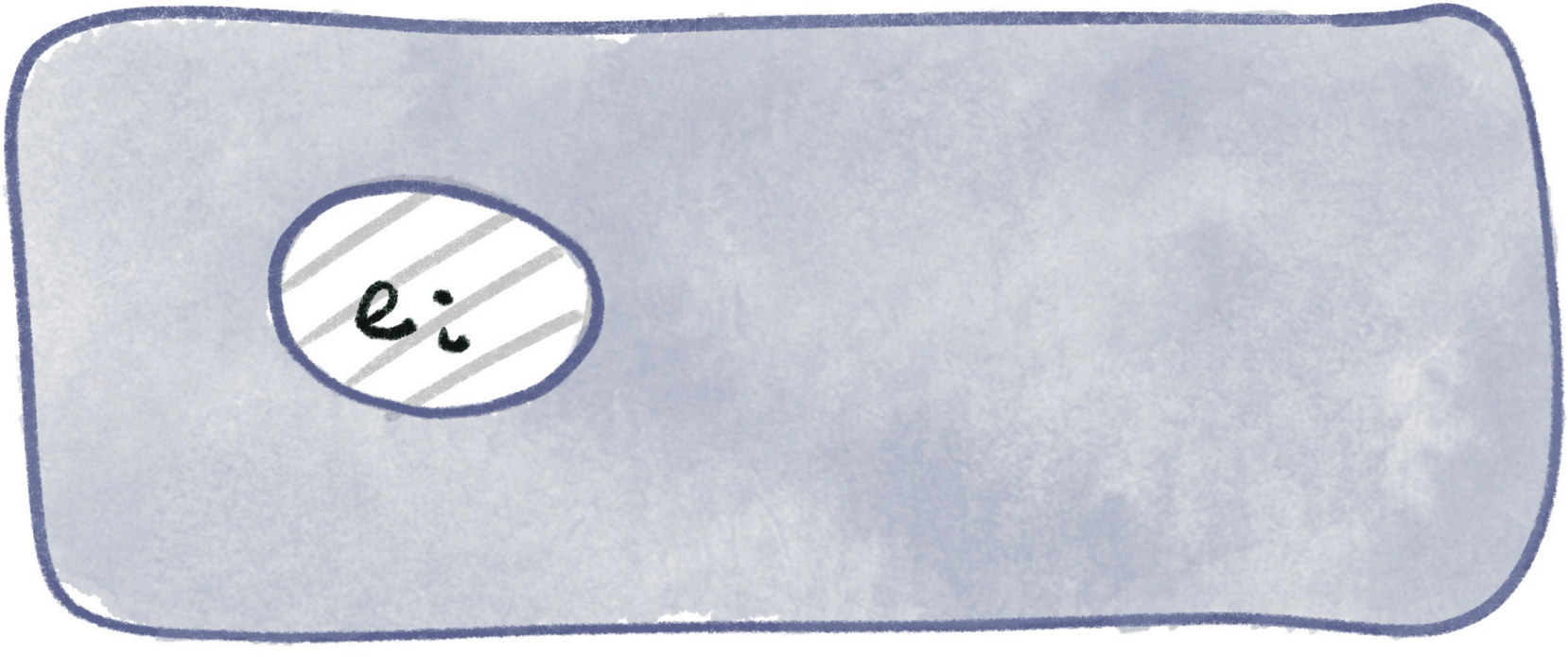
When should we use proof-based generalization?

Then:

- **Apply what you learned:** They generated a new conjecture: A δ-dense subset contains a three-term AP where the difference $e_i + e_j + e_k$ between points is given by **3** standard basis vectors.

Note that after learning from failure, mathematicians quite often conjecture the strongest thing left that *could* be true.

Possible expressions for the difference d



Possible expressions for the difference d

When should we use proof-based generalization?

This autogeneralization process to find the refined cap-set conjecture was not done explicitly, but **instinctively**.

This account is a **best-guess reconstruction** of what was going on (fleetingly) in these mathematicians' minds.

When should we use proof-based generalization?

And so we end up, via (human-implemented) proof-based generalization, at a mathematical research conjecture:

*For any particular density $\delta \geq 0$, we can always find a dimension n big enough so that any δ-dense subset of F_3^n will contain a line such that the difference vector $d = e_i + e_j + e_k$ between points is made of the sum of **3** standard basis vectors.*

When should we use proof-based generalization?

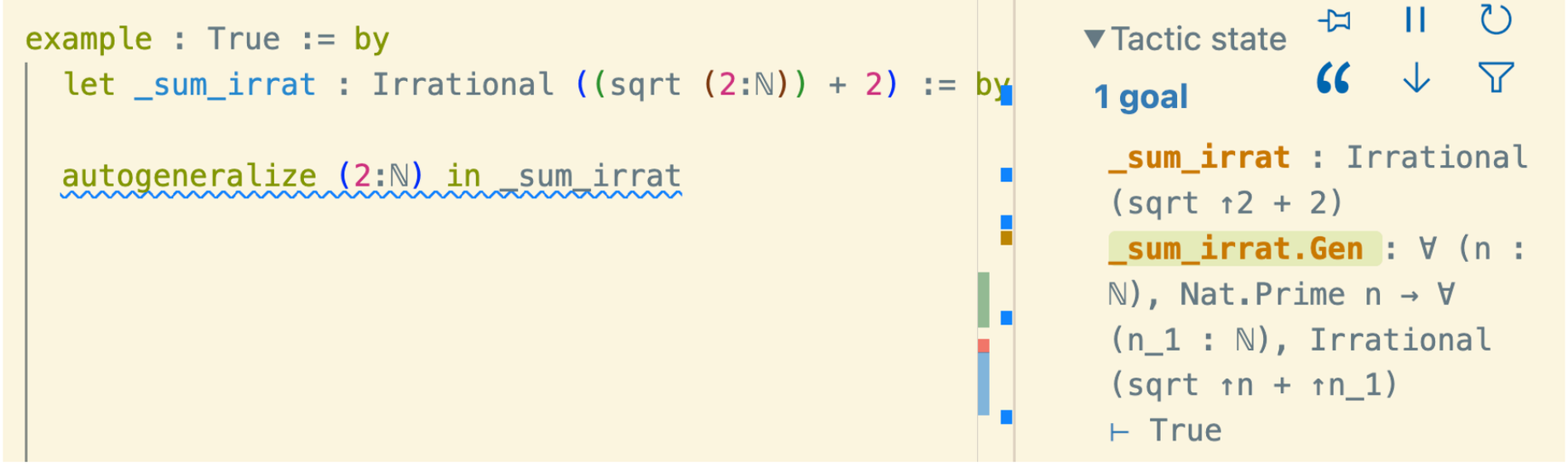
These `autogeneralizations` typically are performed by mathematicians subconsciously, without awareness of any algorithm or heuristic.

Proof-based generalization just *happens* while mathematicians do math.

So there seems to be something natural about proof-based generalization, and if we want to create a human-style theorem prover, we will likely need to incorporate it.

Conclusion

We've designed and implemented an algorithm to improve robustness of **proof-based generalization**….



… which has potential in **enabling automated theorem provers to better learn from failure** when finding proofs **in a human-like way**.