Equality

Equality, type judgments, propositions, and more


Contents
Equality
Type judgments and inference
Propositions as types
Automation
Equality Properties

Propositions and Proofs

- We've previously seen propositions (`def fourIsEven := isEven 4`)
- Proofs have the *types* of the propositions they prove
- For each type of proof, we typically have
  - introduction rules that introduce or construct proofs of that particular type
  - elimination rules that extract information from existing proofs of that type

Equality

Proof that 0 = 0

- Could create an axiom that 0 = 0, but…
  - Axioms are a last resort
  - We would need many axioms (1 = 1, Fido = Fido, etc.)
- A general inference rule is better
  - Rule should be valid for any type, and any value of that type:

$$\frac{\{T : Type\}, (t : T)}{eq\ t\ t}\,(eq.refl)$$

  - Read this "For any type T (inferred), and for any value t (provided) of type T, return a proof of the proposition $eq\ t\ t$.
- eq.refl is an *introduction* rule for equality

Exercises

- Using the type inference rule just discussed:

$$\frac{\{T : Type\}, (t : T)}{eq\ t\ t}\,(eq.refl)$$

  Give an expression in which $eq.refl$ is applied to one argument to derive a proof of $0 = 0$.

- Why exactly can this rule never be used to derive a proof of the proposition that $0 = 1$?

When can we not say x = x?

- The IEEE 754 floating-point standard defines not-a-number as NaN, and specifies NaN ≠ x for all x, including when x is NaN
  - https://en.wikipedia.org/wiki/NaN#Comparison_with_NaN
- In Lean, two proofs might be “assigned” the rfl strategy, but that does not mean they are equal:

```
lemma pf0eq0 : 0 = 0 := rfl
lemma pf1eq1 : 1 = 1 := rfl

lemma pf_rfl_eq_rfl : pf0eq0 = pf1eq1 := rfl
```

Substitution

- Concept: if *a* and *b* are the same value, then *P(a)* and *P(b)* should be the same value
- Textual inference rule

$$\frac{\{T \ : \ Type\}\{P \ : \ T \to Prop\}\{a \ b \ : \ T\}(aeqb \ : \ a = b)(pa \ : \ P \ a)}{P \ b} \ eq.subst$$

- Substitution is an *elimination* rule — what does that mean?

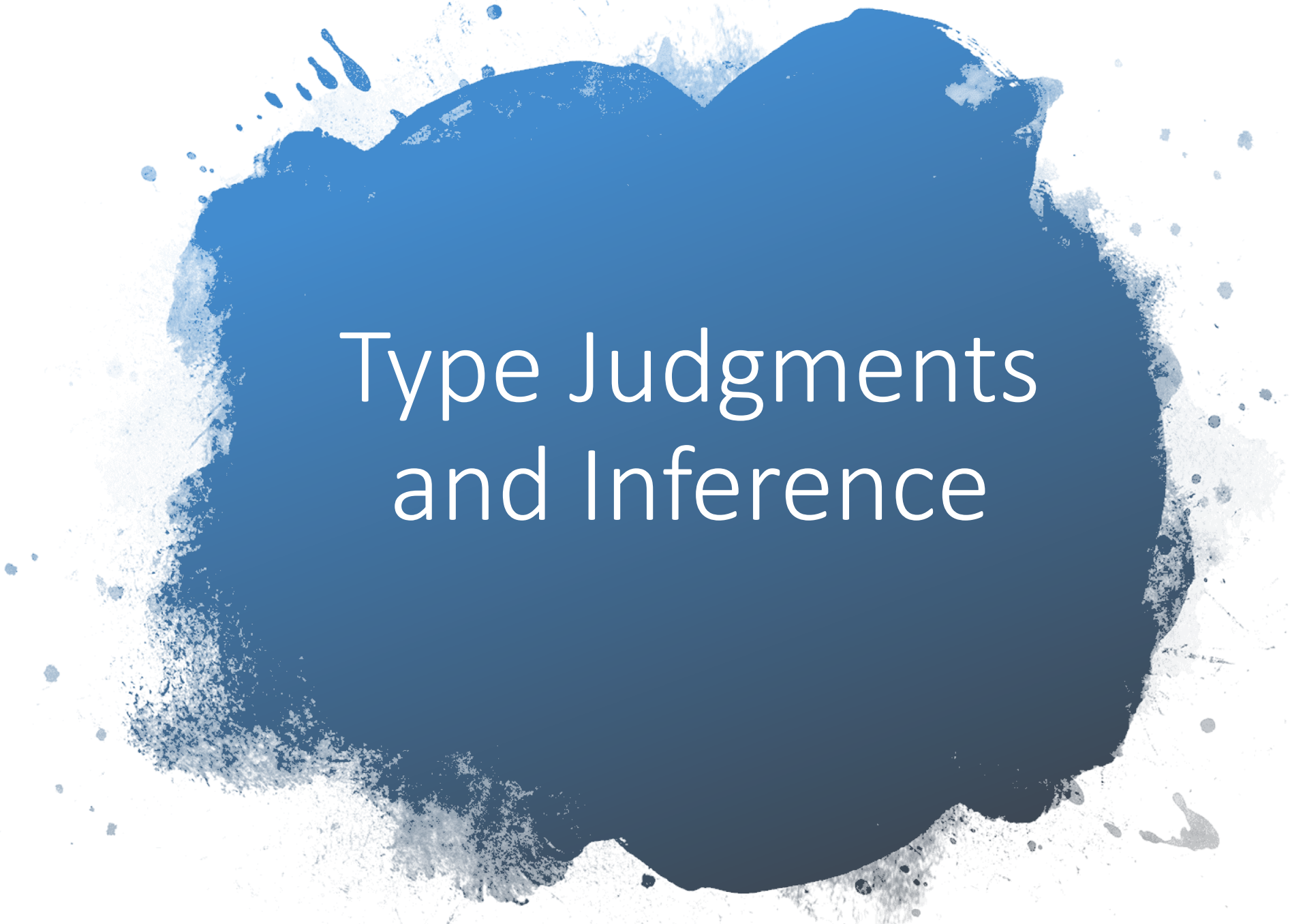
Exercises

- Using the type inference rule just discussed:

$$\frac{\{T \ : \ Type\}\{P \ : \ T \to Prop\}\{a \ b \ : \ T\}(aeqb \ : \ a = b)(pa \ : \ P \ a)}{P \ b} \ eq.subst$$

  and the function f that maps $\mathbb{N}$ to $\mathbb{N}$, give an expression in which $eq.subst$ is applied to one argument to derive a proof of $(f\ 2) \to (f\ 2)$.

- Besides the definition of *f*, what other information do we have to provide to the inference rule?


Type Judgments
and Inference

How Type Judgments Arise

- Consider the inference rule just discussed:

$$\frac{\{T : Type\}, (t : T)}{eq\ t\ t}\,(eq.refl)$$

- In order to use this rule, we lean *infer* the type T.
  - To see what type lean thinks something is, use `#check`

Do Types Have Types?

- What happens if we `#check nat` or `#check bool`, etc?
  - Types are of type `Type`
  - With one *group* of exceptions:
    - `Type` is of type `Type 1`
    - `Type 1` is of type `Type 2`
    - Etc.
  - Note that `Type` is synonymous with `Type 0`.

The Logical Inference Rule eq.refl

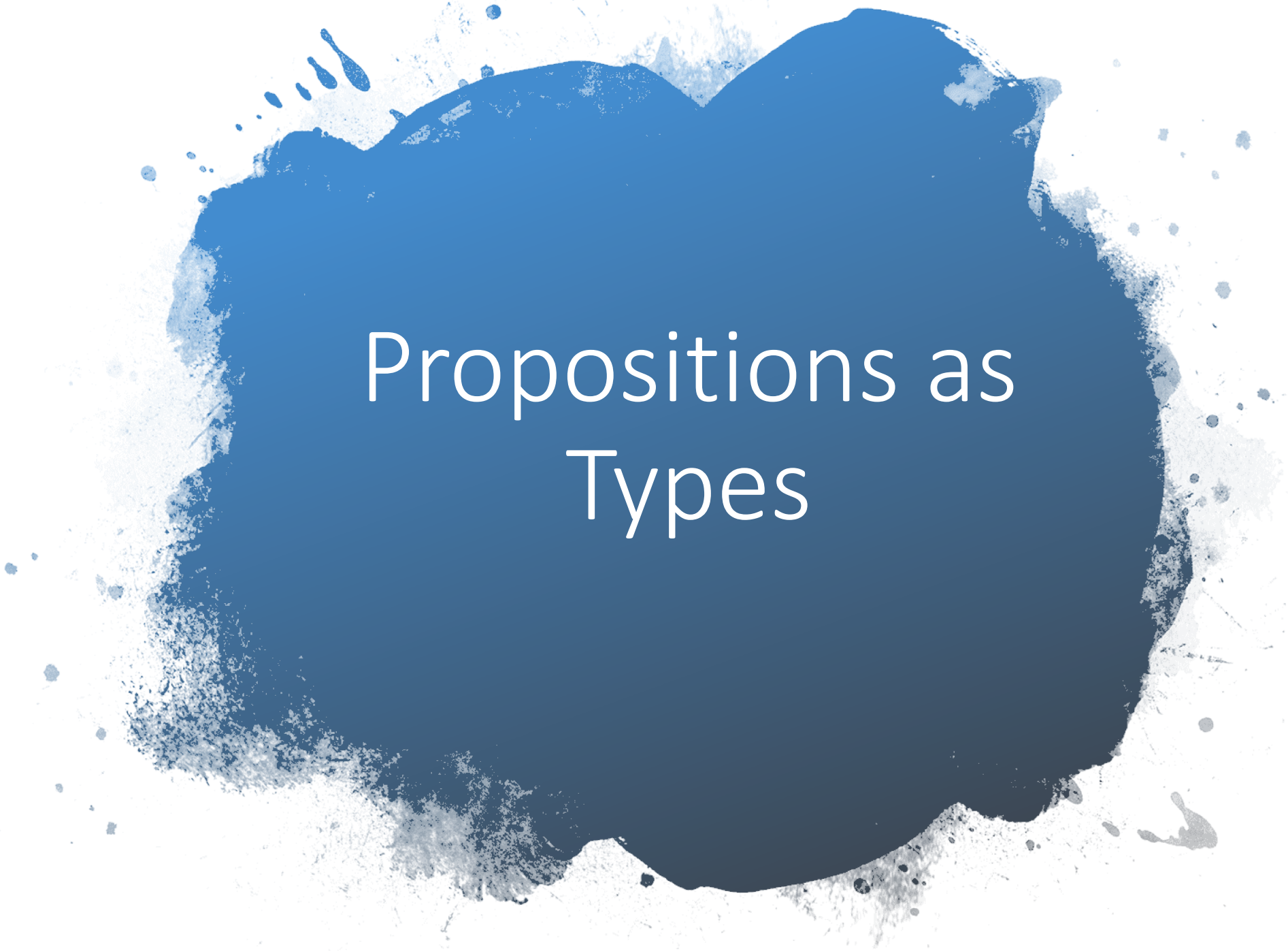
- If we want to prove the proposition that `0 = 0`, we can use either `rfl`, or `eq.refl`.
  - The inference rule `rfl` takes no arguments — lean would infer both the type `nat` for T and the value `0` for t.
  - The inference rule `eq.refl` takes a single argument, the value `0` in this case — lean infers the type `nat` for T.

Checking Lean's Type Inference

- To check what type Lean will infer for a particular value, use `#check`
  - See 01_Equality/03_type_inference.lean
- If we `#check (eq.refl 0)` we see that its *type* is a proof that `0 = 0`!
  - What does this mean?

Type Inference Exercises

- Use `#check` to see what the type is for `eq.refl` *t*, for the case when *t* = `tt` and for the case when *t* = `"Hello Lean!"`
- For the case when *t* = `tt`, what value does Lean infer for the parameter, T?


Propositions as
Types

Propositions and Proofs

- Recall that `#check (eq.refl 0)` we see that its *type* is reported as `0 = 0`!
- Recall that `0 = 0` is a proposition.
- What does it mean when that is reported as a type?
  - These are the reported *types* of inference rules.
  - Lean is reporting that the inference rule is of a proof *type* that satisfies the proposition `0 = 0`.

Types in General

- Recall that the type of 0 and 1 are of type $\mathbb{N}$ in Lean
- Other languages that rely on type inference might consider these to be of type $\mathbb{Z}$
  - Or more precisely of type IEEE 32-bit or 64-bit integers (signed or unsigned).
- Many programming languages do type-checking, but how rigorous is that type-checking?
- The maximum possible 32-bit unsigned integer is 4,294,967,295
  - What is the type of 4,294,967,295 + 1?
- In SPARK Ada, it's not a 32-bit unsigned integer, and if you will be unable to do perform that computation and assign it to a 32-bit integer. SPARK Ada will type-check this at compile time!
- Why does this matter?

Boeing 787 Dreamliner Integer Overflow

- From Ars Technica: Boeing 787 Dreamliners contain a potentially catastrophic software bug
- “[A] Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode”
- If this code had been written in SPARK Ada (or with a similarly strong type-checker), this defect would never have gotten onto the plane
- This is one reason why Discrete Math is so important to Computer Science!

Binding Types and Values to Variables in Lean

- To declare a variable n of type $\mathbb{N}$ and with the value 1 in Lean:

```
def n       : nat  :=    1
-- variable  type bind  value
```

- We can also create a variable `p` of type `0 = 0` (a proposition) and assign it a value of `eq.refl 0` (a proof):

```
def p       : 0 = 0 :=   (eq.refl 0)
-- variable  type  bind  value
```

- Because this is a proposition, however, `lemma` (or `theorem`) is preferred:

```
lemma p  : 0 = 0 :=   (eq.refl 0)
```

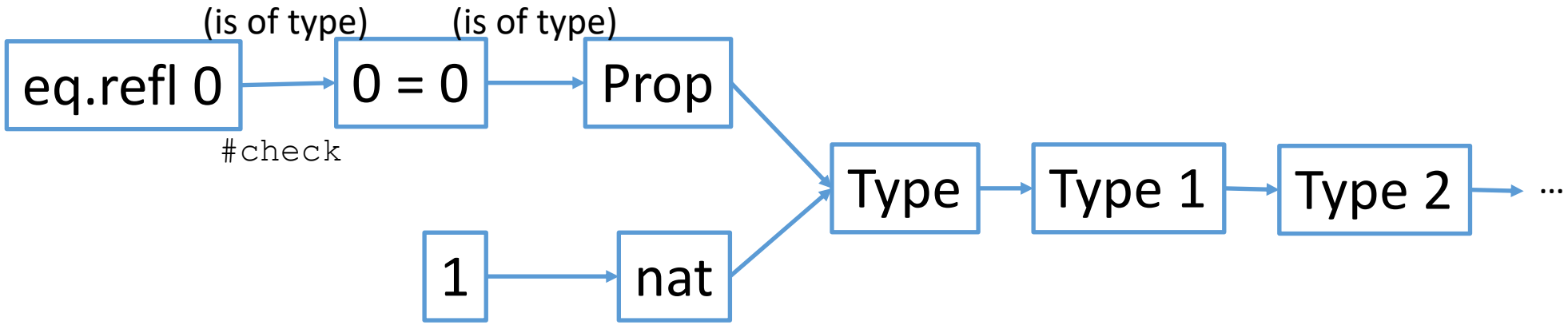
Exercises

- To the variable s, bind a proof of the proposition that "Hello Lean!" is equal to itself.
- To the variable s, bind a proof of the proposition that tt is equal to itself.
- Explain precisely why Lean reports an error for this code and what it means:
  ```
  def p' : 0 = 0 := (eq.refl 1)
  ```
- Use `theorem` instead of def to bind a proof of 0 = 0 to a variable, s'.

More Exercises

- Prove theorems for the following propositions:
  - oeqo : 1 = 1
    - lemma oeqo : 1 = 1 := eq.refl 1
  - heqh : "Hello Lean!" = “Hello Lean!”
    - lemma heqh : “Hello Lean!” = “Hello Lean!” := eq.refl “Hello Lean!”
  - teqt : 2 = 1 + 1
    - lemma teqt : 2 = 1 + 1 := eq.refl (3 - 1)
- What is the type of `(0 = 1)` in Lean? Answer before you `#check` it, then `#check` it to confirm.
- What is the type of `"Hello Lean" = "Hello Lean"` in Lean?

Proofs and Propositions as Types in Lean



Automation

Tactics/Strategies

- The inference rule `eq.refl` is very simple — it takes a single argument, e.g. `0`, and creates a proof of a proposition that the argument equals itself, e.g., `0 = 0`
- The tactic `rfl` might seem simpler because it takes no argument, but because it must infer not just the *type* but also the *value* to use for the proposition, the tactic's logic is more complicated
- Building up more complicated rules from simpler rules is common in provers
  - In some provers (e.g., Lean) these are called *tactics*, and in other provers (e.g., PVS) these are call *strategies*, but the idea is the same across provers

Reduction in Lean

- Another useful feature of Lean is its ability to automatically *reduce* simple arithmetic examples
  - Use #reduce to see how Lean will reduce
  - This is useful for proofs, e.g., when proving that 2 = 1 + 1:
    ```
    theorem t1 : 2 = 1 + 1 := rfl
    ```
  - Lean can *reduce* 1+1 to 2 and then use rfl to prove 2 = 2.

Exercises

- Use rfl to produce a proof, h, of the proposition, "Hello" = "He" ++ "llo".
  - lemma h : "Hello" = "He" ++ "llo" := rfl
- Use rfl to prove p: 3*3+4*4=5*5.
  - lemma p: 3 * 3 + 4 * 4 = 5 * 5 := rfl
- Prove as a theorem, tthof (a silly and uninformative name to be sure), that 2 + 3 = 1 + 4.
  - lemma tthof : 2 + 3 = 1 + 4 := rfl
- Prove as a theorem, hpleqhl, that "Hello " ++ "Lean!" is equal to "Hello Lean!”
  - lemma hpleqhl := "Hello " ++ "Lean!" = "Hello Lean!" := rfl

Equality Properties

Properties of Equality

- Reflexivity
  - $a = a$
- Symmetry
  - $a = b \Leftrightarrow b = a$
- Transitivity
  - $a = b \wedge b = c \Rightarrow a = c$
- These three properties are properties of *equivalence*

Exercise

- Parity is whether an integer is even or odd. Let us define a relationship “sameParity” as being whether two integers have the same parity (i.e., they are both even or both odd)

```
def sameParity(a b: ℕ): Prop := (a % 2) = (b % 2)
```

  - Is this relationship reflexive? I.e., is the following proposition true?

```
∀(x: ℕ), (sameParity x x)
```

  - Is the relationship symmetric? I.e., is the following proposition true?

```
∀(x y: ℕ), (sameParity x y) ↔ (sameParity y x)
```

  - Is the relationship transitive? I.e., is the following proposition true?

```
∀(x y z: ℕ), (sameParity x y) ∧ (sameParity y z) →
  (sameParity x z)
```

  - Is the relationship an *equivalence* relationship?

Fin