



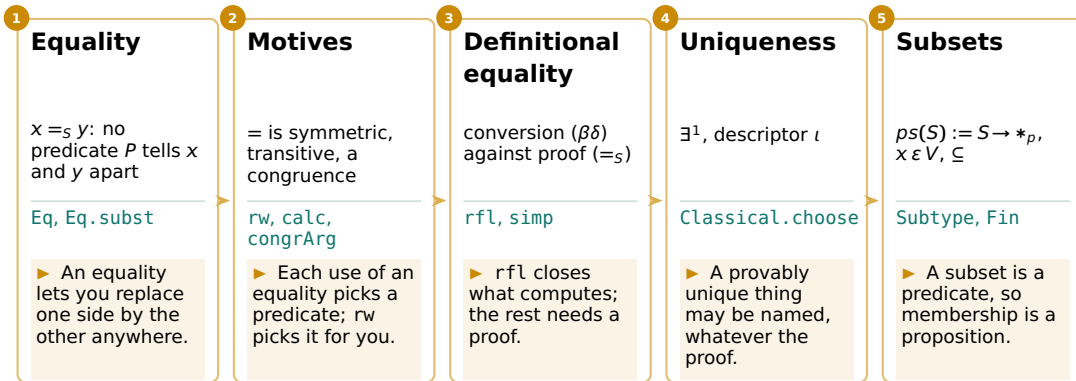
Computational Logic in the Era of AI

Equality, and Sets as Predicates

Sergey Mechtaev
mechtaev@pku.edu.cn

Term 1 • 26–27

This lecture



History: what it means to be equal



G. W. Leibniz

1646–1716

Equal means
indiscernible: no
predicate tells the
two apart.

Leibniz's idea becomes $=_S$, Frege's ι . Photos: Wikimedia.

1686 Leibniz: things are the same when one may be substituted for the other everywhere, *salva veritate*.

1879 Frege's *Begriffsschrift*: the identity sign between names, with substitution as an axiom; later "the *F*", a definite description.

1908 Zermelo: a subset is cut out of a set by a predicate, the axiom of separation.

1972 Martin-Löf: equality as a type with one constructor and an elimination rule. Lean's `Eq`.



Gottlob Frege

1848–1925

ι names the unique
thing with a
property.

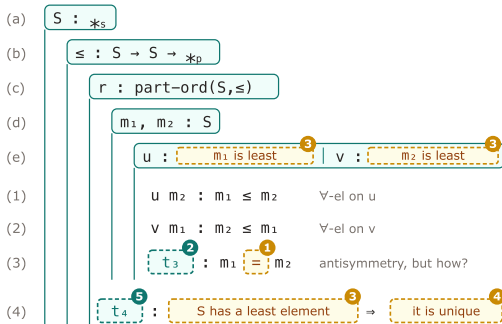
N&G §12.2, §12.7, §13.1

A first piece of mathematics

Lemma. If S is partially ordered by $\leq$ and has a least element, that element is unique.

Proof. Let m_1, m_2 be least: $m_1 \leq m_2$ and $m_2 \leq m_1$, so $m_1 = m_2$ by antisymmetry. $\square$

the proof as a flag derivation



what is missing, and what supplies it

- 1 the symbol $=$
 $\rightarrow$ a definition: Leibniz equality
- 2 a proof t_3 of $m_1 = m_2$
 $\rightarrow$ substitutivity, a theorem about $=$
- 3 "least", "has a least element"
 $\rightarrow$ a definition Least, and $\exists$
- 4 "unique"
 $\rightarrow \exists^1$, exactly one
- 5 a proof t_4 of the whole lemma
 $\rightarrow$ all of the above

missing proposition

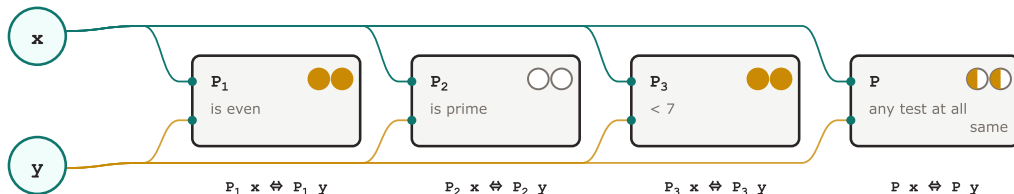
missing proof object

Each numbered gap is filled later; its slide carries the badge.

N&G §12.1, Fig. 12.1

What does $x = y$ mean?

Leibniz: x and y are equal when no predicate P tells them apart.



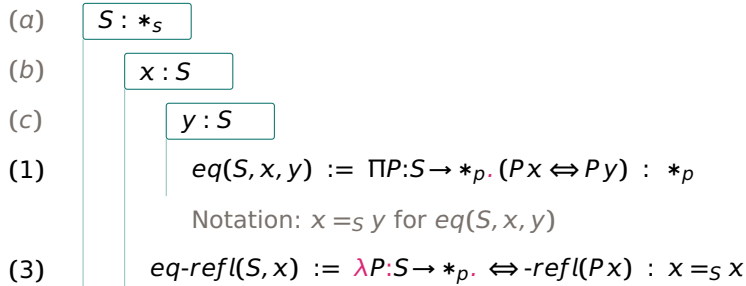
$\text{eq}(S, x, y) := \prod P : S \rightarrow *_p . (P x \Leftrightarrow P y)$ — indiscernible by every predicate

Second-order: the $\prod$ ranges over predicates, not elements.

Not an axiom but a definition, so the properties of equality are theorems.

N&G §12.2

Equality as a definition



- Second order: P ranges over predicates, $S \rightarrow *_p : \square$.
- Reflexivity is one $\lambda : \Leftrightarrow-refl(A) := \wedge-in(\lambda a : A. a, \lambda a : A. a) : A \Leftrightarrow A$.
- Even $\Rightarrow$ in place of $\Leftrightarrow$ would do.

N&G §12.2, Figs. 12.2–12.3, Rem. 12.2.1–12.2.2

Leibniz equality, written out

```
-- eq(S,x,y) :=  $\prod P : S \rightarrow *_{\mathcal{P}}. (P\ x \Leftrightarrow P\ y)$ 
-- "x and y are equal iff they are INDISCERNIBLE by any predicate."
def LEq {S : Type} (x y : S) : Prop :=  $\forall P : S \rightarrow \text{Prop}, P\ x \leftrightarrow P\ y$ 

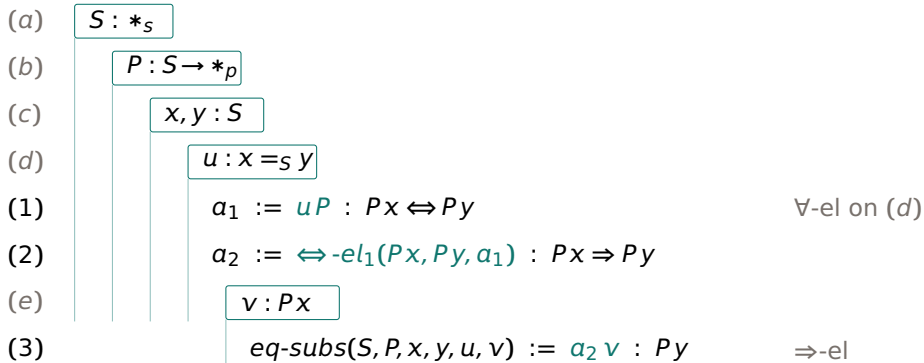
-- reflexivity: for every P,  $\Leftrightarrow$ -refl, which Lean calls Iff.rfl
theorem leq_refl {S : Type} (x : S) : LEq x x := fun _ => Iff.rfl

--  $\Rightarrow$  instead of  $\Leftrightarrow$  would already do:  $P\ x \Rightarrow P\ y$  for all P gives the converse.
def LEq' {S : Type} (x y : S) : Prop :=  $\forall P : S \rightarrow \text{Prop}, P\ x \rightarrow P\ y$ 

-- The  $\prod$  ranges over PREDICATES,  $P : S \rightarrow *_{\mathcal{P}}$ , and  $S \rightarrow *_{\mathcal{P}} : \square$ . So this is
-- a SECOND-ORDER definition, not a first-order  $\forall$  over elements.
#check @LEq      -- @LEq : {S : Type}  $\rightarrow S \rightarrow S \rightarrow \text{Prop}$ 
```

Substitutivity: the rule for using an equality

Substitute equals for equals: if $x =_S y$ and Px , then Py , for every predicate P .



Using an equality means choosing P .

N&G §12.2, Fig. 12.4

What Lean uses instead

```
-- Lean does not use Leibniz's definition. 'Eq' is an inductive with ONE
-- constructor, and its recursor is what substitutivity becomes.
#print Eq
-- inductive Eq.{u_1} : {α : Sort u_1} → α → α → Prop
-- number of parameters: 2
-- constructors: Eq.refl : ∀ {α : Sort u_1} (a : α), a = a

#check @Eq.subst
-- ∀ {α : Sort u_1} {motive : α → Prop} {a b : α}, a = b → motive a → motive b
-- Compare eq_subs(S,P,x,y,u,v) : P y. Same shape; the predicate P is
-- what Lean calls the 'motive'.
```

The predicate P is what Lean calls the motive. Why `Eq.subst` is the recursor of `Eq`: a later lecture.

The two agree

*-- The two notions coincide. One direction substitutes along h,
-- starting from reflexivity ...*

```
theorem eq_to_leq {S : Type} {x y : S} (h : x = y) : LEq x y :=  
  fun P => h ▸ Iff.rfl
```

*-- ... the other instantiates the quantifier at the predicate (x = .):
-- once more, the proof is the choice of predicate.*

```
theorem leq_to_eq {S : Type} {x y : S} (h : LEq x y) : x = y :=  
  (h (fun z => x = z)).mp rfl
```

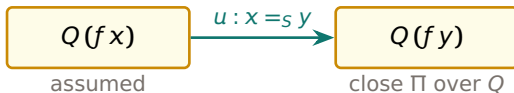
One direction is substitution with the motive $\lambda z. (x = z)$, and that motive is the whole proof.

Congruence, proved twice

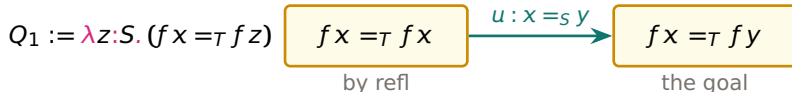
Congruence. If $u : x =_S y$ and $f : S \rightarrow T$, then $f x =_T f y$. Two proofs, two choices of predicate:

first proof

$\lambda z : S. Q(f z)$,
for any $Q : T \rightarrow *_{\mathcal{P}}$



second proof

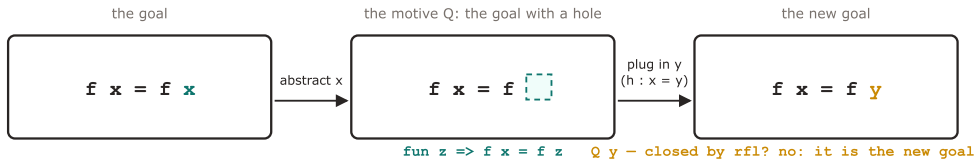


The second is one line: $eq\text{-}cong_2 := eq\text{-}subs(S, Q_1, x, y, u, eq\text{-}refl(T, f x))$.

N&G §12.3, Figs. 12.5–12.6

Choosing Q is choosing the motive

To use $x =_S y$, say which occurrences of x become the hole. That predicate is Lean's motive.



N&G §12.3, §12.5

“motive is not type correct”

```
-- 'rw' builds the MOTIVE Q by abstracting 'n' out of the goal, then checks
-- it. Here the goal mentions 'v : Fin (n+1)' (a natural below n+1, bundled
-- with that proof), whose TYPE mentions n -- so the abstraction is ill-typed:
#guard_msgs(drop error) in
example (n m : Nat) (v : Fin (n + 1)) (h : n = m) : v.val < n + 1 := by
  rw [h]
-- Tactic 'rewrite' failed: motive is not type correct:
--   fun _a => ↑v < _a + 1
-- Error: Application type mismatch: The argument v has type
--   Fin (n + 1) but is expected to have type Fin (_a + 1) ...

-- Two cures: 'subst' replaces n EVERYWHERE, v's type included; '▷' takes
-- the motive from the proof term's type instead of from the goal.
example (n m : Nat) (v : Fin (n + 1)) (h : n = m) : v.val < n + 1 := by
  subst h; exact v.isLt
example (n m : Nat) (v : Fin (n + 1)) (h : n = m) : v.val < n + 1 :=
  h ▷ v.isLt
```

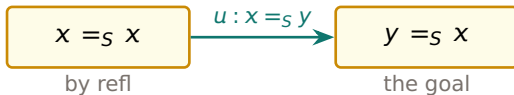
No predicate Q with Qn the goal typechecks here: “choose Q ” is a real error message.

Symmetry and transitivity, from substitutivity

Two more choices of predicate, with $u : x =_S y$ and $v : y =_S z$:

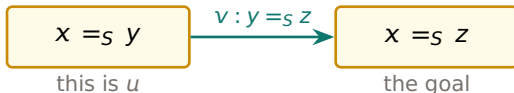
symmetry

$Q_2 := \lambda z:S. (z =_S x)$



transitivity

$Q_3 := \lambda w:S. (x =_S w)$



$eq\text{-}sym(S, x, y, u) := eq\text{-}subs(S, Q_2, x, y, u, eq\text{-}refl(S, x)) : y =_S x$

$eq\text{-}trans(S, x, y, z, u, v) := eq\text{-}subs(S, Q_3, y, z, v, u) : x =_S z$

$Q_2 x =_\beta x =_S x$: Lean does (*conv*) silently.

N&G §12.5, Figs. 12.10–12.12

Motives in Lean

```
-- Congruence: the motive inferred from the goal, or given.
theorem cong' {S T : Type} (f : S → T) {x y : S} (h : x = y) : f x = f y :=
  h ▸ rfl -- motive inferred
theorem cong'' {S T : Type} (f : S → T) {x y : S} (h : x = y) : f x = f y :=
  @Eq.subst S (fun z => f x = f z) x y h rfl -- motive Q1

-- Symmetry and transitivity, with Q2 and Q3 given.
theorem sym' {S : Type} {x y : S} (h : x = y) : y = x :=
  @Eq.subst S (fun z => z = x) x y h rfl -- Q2 x is x = x
theorem trans' {S : Type} {x y z : S} (h : x = y) (k : y = z) : x = z :=
  @Eq.subst S (fun w => x = w) y z k h -- Q3 y is x = y

-- In the library: 'congrArg', 'Eq.symm', 'Eq.trans'.
```

rw chooses the motive for you

```
-- 'rw [h]' is eq-subs with the predicate chosen FOR you: it abstracts every
-- occurrence of h's left side out of the goal, and that abstraction is Q.
theorem rw1 (a b : Nat) (h : a = b) (f : Nat → Nat) : f a + a = f b + b := by
  rw [h] -- Q := fun z => f z + z = f b + b; the rest is Eq.refl

example (a b : Nat) (h : a = b) (k : b = 0) : a = 0 := by
  rw [h, k] -- one equation after another, each left to right
example (a b : Nat) (h : b = a) : a = b := by
  rw [← h] -- right to left
example (a b : Nat) (h : a = b) (k : a + 1 = 5) : b + 1 = 5 := by
  rw [h] at k; exact k -- in a hypothesis

#print rw1
-- fun a b h f => Eq.mpr (id (congrArg (fun _a => f _a + _a = f b + b) h))
-- (Eq.refl (f b + b))
```

Read the term: `congrArg` at the motive is congruence, `Eq.mpr` is substitution along it, and `Eq.refl` closes what is left. All of the equality theory so far is inside one `rw`.

Chains of steps: calc

```
-- 'calc' chains equalities step by step. Each step names its
-- justification; transitivity is applied for you.
def d (n : Nat) : Nat := n + n

example (a b : Nat) (h : a = b) (k : b = 5) : d a = 10 :=
  calc d a = d b := by rw [h]      -- congruence
    _      = d 5 := by rw [k]
    _      = 10  := rfl            -- definitional:  $\delta$ , then arithmetic

-- Mixed relations chain too; the result is the strongest one used.
example (a b c e : Nat) (h1 : a ≤ b) (h2 : b < c) (h3 : c ≤ e) : a < e :=
  calc a ≤ b := h1
    _ < c := h2
    _ ≤ e := h3
```

Orders

“Partial order” is five definitions in one context:

- (a) $S : *_S$
- (b) $\leq : S \rightarrow S \rightarrow *_p$
 - Notation: $x \leq_S y$ for $\leq x y$
 - (1) $refl(S, \leq) := \forall x:S. (x \leq_S x) : *_p$
 - (2) $trans(S, \leq) := \forall x, y, z:S. (x \leq_S y \Rightarrow y \leq_S z \Rightarrow x \leq_S z) : *_p$
 - (3) $pre\text{-}ord(S, \leq) := refl(S, \leq) \wedge trans(S, \leq) : *_p$
 - (4) $antisymm(S, \leq) := \forall x, y:S. (x \leq_S y \Rightarrow y \leq_S x \Rightarrow x =_S y) : *_p$
 - (5) $part\text{-}ord(S, \leq) := pre\text{-}ord(S, \leq) \wedge antisymm(S, \leq) : *_p$

3 $Least(S, \leq, m) := \forall n:S. (m \leq_S n).$

N&G §12.4, Fig. 12.7, Rem. 12.4.1

A skeleton proof: the types, and nothing else

First the flags and the types of the lines; every proof is still a hole.

skeleton proof: only the types

```
(a) S : *S
(b) ≤ : S → S → *p
(c) r : part-ord(S, ≤)
(d) m1, m2 : S
(e) u : ∀n. m1 ≤ n | v : ∀n. m2 ≤ n

(1) ... : m1 ≤ m2
(2) ... : m2 ≤ m1
(4) ... : antisymm(S, ≤)
(6) ... : m1 ≤ m2 → m2 ≤ m1 → m1 = m2
(8) ... : m1 = m2
```

the same skeleton, as a script

```
-- in context: the five flags (a)–(e)
-- S, le, r, m1 m2, u, v

have h1 : m1 ≤ m2 := ?_
have h2 : m2 ≤ m1 := ?_
have anti : Antisymm le := ?_
have step : m1 ≤ m2 → m2 ≤ m1 → m1 = m2 := ?_
show m1 = m2
```

the gold dots are the holes; the filled proof fills them, and exact/rw fill them in Lean

Filling the holes fills **2**.

N&G §12.4, Fig. 12.8

The proof objects, filled in

- (a) $S : *_S \mid \leq : S \rightarrow S \rightarrow *_p \mid r : \text{part-ord}(S, \leq)$
- (d) $m_1, m_2 : S$
- (e) $u : \forall n : S. (m_1 \leq_S n) \mid v : \forall n : S. (m_2 \leq_S n)$
- (1) $a_1 := u\ m_2 : m_1 \leq_S m_2$ $\forall$ -el
- (2) $a_2 := v\ m_1 : m_2 \leq_S m_1$ $\forall$ -el
- (4) $a_4 := \wedge\text{-el}_2(\dots, r) : \text{antisymm}(S, \leq)$ $\wedge$ -el₂, then (conv)
- (6) $a_6 := a_4\ m_1\ m_2 : m_1 \leq_S m_2 \Rightarrow m_2 \leq_S m_1 \Rightarrow m_1 =_S m_2$ $\forall$ -el twice
- (8) $a_8 := a_6\ a_1\ a_2 : m_1 =_S m_2$ $\Rightarrow$ -el twice
- (9) $a_9(S, \leq, r) := \lambda m_1 m_2 : S. \lambda u. \lambda v. a_8 : \forall m_1, m_2 : S. (\dots \Rightarrow m_1 =_S m_2)$ (abst)

2 t_3 is a_8 . Lines (3), (5), (7) only unfold, by (conv).

N&G §12.4, Fig. 12.9

The same proof, as named lines

-- The flag derivation, line for line. A 'have' is a named line; 'exact' is the last.

```
theorem least_unique {S : Type} (le : S → S → Prop) (r : PartOrd le)
  (m1 m2 : S) (u : ∀ n, le m1 n) (v : ∀ n, le m2 n) : m1 = m2 := by
  have a1 : le m1 m2 := u m2           -- (1)  ∀-el on u
  have a2 : le m2 m1 := v m1           -- (2)  ∀-el on v
  have a4 : Antisymm le := r.2         -- (3)-(4)  ∧-el2 on r
  have a6 : le m1 m2 → le m2 m1 → m1 = m2 := a4 m1 m2 -- (5)-(6)
  exact a6 a1 a2                       -- (7)-(8)
```

```
#print least_unique
```

```
-- fun {S} le r m1 m2 u v => have a1 := u m2; have a2 := v m1;
--   have a4 := r.right; have a6 := a4 m1 m2; a6 a1 a2
```

The haves survive in the term as local definitions: the flag derivation, folded into one object.

Two equalities

propositional: $x =_S y$, a proposition; an inhabitant is needed (Eq, rw, calc)

definitional: $M =_{\beta\delta\iota} N$, a judgement;
the kernel decides it (rfl)

- $2 + 2 = 4$
- $d\ 3 = 6$ (δ , then arithmetic)
- $n + 0 = n$ (I on the second argument)

- $0 + n = n$
(needs induction)
- $a + b = b + a$
(needs a lemma)
- $V = W$
(needs an axiom: extensionality)

everything inside the teal region is also in the gold one, by rfl; nothing goes the other way

Teal sits inside gold, not the other way round: that is why rw exists. N&G §4.7, §9.7, §12.2

Definitional and propositional, side by side

	definitional	propositional
what it is	a judgement	a proposition
written	$M =_{\beta\delta} N$	$M =_S N$
comes from	the (<i>conv</i>) rule	a definition
decided by	the kernel	nobody: it needs a proof
closed by	rfl	rw, Eq.subst, calc

$n + 0 = n$ is definitional: addition recurses on its second argument. $0 + n = n$ is not: it needs an induction.

N&G §4.7, §12.2

Where rfl stops, and what takes over

-- 'Nat.add' recurses on its SECOND argument, so one of these computes and the other is stuck on the variable n:

```
example (n : Nat) : n + 0 = n := rfl
```

```
#guard_msgs(drop error) in
```

```
example (n : Nat) : 0 + n = n := rfl
```

-- The type of 'Eq.refl ?a' ... is not definitionally equal to '0 + n = n'

-- What closes it is a theorem from Δ -- proved by induction, next lecture:

```
example (n : Nat) : 0 + n = n := Nat.zero_add n
```

-- With n closed, computation is back, and 'decide' runs it in the kernel:

```
example : 0 + 5 = 5 := by decide
```

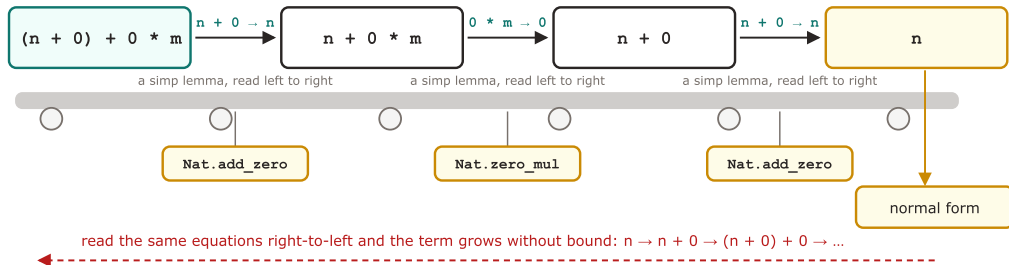
Past the boundary: a lemma, a closed instance the kernel runs, or rewriting to a normal form.

Rewriting to a normal form

A rewrite rule is an equation read one way, $s \rightarrow t$. `simp` rewrites with every lemma tagged `@[simp]`: its simp set.

Orientation is the whole design decision.

confluence and termination are theorems for $\beta\delta$; for your simp set they are your obligation.



Unique existence

4 Three grades of “exists”, for $P : S \rightarrow *p$:

	definition	reads	e.g.
$\exists^{\geq 1}(S, P)$	$\exists x:S. P x$	at least one	○ ● ○ ● ●
$\exists^{\leq 1}(S, P)$	$\forall y, z:S. (P y \Rightarrow P z \Rightarrow y =_S z)$	at most one	○ ○ ○ ○ ○ ○ ○ ● ○ ○
$\exists^1(S, P)$	$\exists^{\geq 1}(S, P) \wedge \exists^{\leq 1}(S, P)$	exactly one	○ ○ ● ○ ○

● an element with P

5 Line (9) of the proof is $\exists^{\leq 1}$ for $P := \text{Least}$; with $w : \exists^{\geq 1}$, $\wedge$ -in gives $\exists^1$.

Theorems and proofs

If $\Gamma \vdash a(\bar{x}) := N : *p$ and $\Gamma \vdash b(\bar{x}) := M : N$, then $a(\bar{x})$ is a theorem and $b(\bar{x})$ its proof.

N&G §12.6, Figs. 12.14–12.15

The descriptor ι

What is unique can be named: $\iota_{x:S}(Px)$, the x with Px . Two primitive definitions, the object and its property:

(a)	$S : *_{\mathcal{S}} \mid P : S \rightarrow *_{\mathcal{P}} \mid u : \exists^1 x:S. Px$	
(1)	$\iota(S, P, u) := \perp\!\!\!\perp : S$ Notation: $\iota_{x:S}^u(Px)$ for $\iota(S, P, u)$	no body
(2)	$\iota\text{-prop}(S, P, u) := \perp\!\!\!\perp : P(\iota_{x:S}^u(Px))$	no body

- Whatever has P is the ι : $\forall z:S. (Pz \Rightarrow z =_S \iota_{x:S}^u(Px))$.
- So the proof u does not matter: $\iota(S, P, u_1) =_S \iota(S, P, u_2)$.
- The minimum: $Min := \iota(S, \lambda m. Least(S, \leq, m), \dots)$.

N&G §12.7, Figs. 12.16–12.18, Rem. 12.7.2

Classical.choose is ι

```
--  $\exists^1$  is "at least one" and "at most one". Core Lean has no  $\exists^1$ , so the
-- definition is written out.
def ExUnique {S : Type} (P : S → Prop) : Prop :=
  ( $\exists$  x, P x)  $\wedge$   $\forall$  y z, P y → P z → y = z

--  $\iota(S, P, u) := \bot\bot : S$  and  $\iota\text{-prop}(S, P, u) := \bot\bot : P (\iota \dots)$ .
-- Lean's are 'Classical.choose' and 'choose_spec', built on an axiom:
noncomputable def iota {S : Type} (P : S → Prop) (u : ExUnique P) : S :=
  Classical.choose u.1
theorem iota_prop {S : Type} (P : S → Prop) (u : ExUnique P) : P (iota P u) :=
  Classical.choose_spec u.1
#print axioms iota_prop      -- 'iota_prop' depends on axioms: [Classical.choice]

-- Whatever satisfies P is the  $\iota$ .
theorem eq_iota {S : Type} (P : S → Prop) (u : ExUnique P) (z : S) (v : P z) :
  z = iota P u := u.2 z _ v (iota_prop P u)

noncomputable: a primitive constant has no body to run, and the axiom audit names it.
```

Irrelevance of proof

*-- λD does not make proof irrelevance a principle. Lean's kernel has it
-- for Prop, definitionally: two proofs of one proposition are rfl-equal.*
`example (p : Prop) (h1 h2 : p) : h1 = h2 := rfl`

*-- So the worry that $\iota(S, P, u_1)$ and $\iota(S, P, u_2)$ might differ
-- cannot even be stated as a difference:*

`example {S : Type} (P : S → Prop) (u1 u2 : ExUnique P) :
iota P u1 = iota P u2 := rfl`

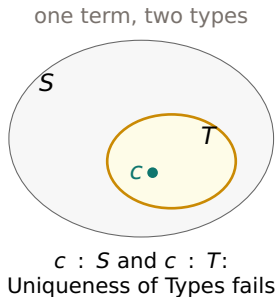
*-- What keeps it consistent: a proof may not be turned into DATA. A Prop's
-- recursor eliminates only into Prop, so no function can tell two
-- proofs apart:*

`#guard_msgs(drop error) in
def theWitness (h : $\exists$ n : Nat, 3 < n) : Nat := match h with | ⟨n, _⟩ => n
-- error: recursor 'Exists.casesOn' can only eliminate into 'Prop'`

λD proves the equation; Lean's kernel makes it a conversion. That is one of the places Lean is not λD .

Why subsets cannot be types

Suppose a subset $T \subseteq S$ were a type, so that $c : T$.



type checking becomes proof search

type-check FM , with
 F the square root

needs $M : \{x \in \mathbb{R} \mid x \geq 0\}$

find a proof of $M \geq 0$

undecidable in general
(Church, Turing 1936)

So typing stays decidable, and subsets need another representation. N&G §13.1

Subsets are predicates

A subset of S is a predicate on S ; predicates are already in λD .

- (a) $S : *_{\mathcal{S}}$
- (1) $ps(S) := S \rightarrow *_{\mathcal{P}} : \square$
- (b) $x : S \mid V : ps(S)$
- (2) $element(S, x, V) := Vx : *_{\mathcal{P}}$ Notation: $x \in V$
- (c) $V, W : ps(S)$
- (3) $\subseteq(S, V, W) := \forall x:S. (x \in V \Rightarrow x \in W) : *_{\mathcal{P}}$
- (4) $\cup(S, V, W) := \lambda x:S. (x \in V \vee x \in W) : ps(S)$

$x \in V$ is a proposition to prove, not a typing $x : V$. N&G §13.2, Figs. 13.1–13.2

$ps(S)$ in Lean

```
-- Lean core has no 'Set'; these lines are all of it.
abbrev PS (S : Type) : Type := S → Prop
def Mem {S : Type} (V : PS S) (x : S) : Prop := V x
instance {S : Type} : Membership S (PS S) := ⟨Mem⟩
def Subs {S : Type} (V W : PS S) : Prop :=
  ∀ x : S, x ∈ V → x ∈ W
infix:50 " ⊆ " => Subs
def IS {S : Type} (V W : PS S) : Prop := V ⊆ W ∧ W ⊆ V -- same elements
def diff {S : Type} (V W : PS S) : PS S := fun x => x ∈ V ∧ ¬ x ∈ W -- V \ W
def compl {S : Type} (V : PS S) : PS S := fun x => ¬ x ∈ V -- V^c

example (S : Type) (V W : PS S) :
  (V ⊆ W) = ∀ x : S, x ∈ V → x ∈ W := rfl -- the guarded ∀, verbatim
```

The instance line tells Lean that $x \in V$ means $\text{Mem } V \ x$. Later slides write $V \subseteq W$ as $V \sqsubseteq W$.

Membership needs a proof

*-- $x \in V$ is not a typing $x : V$ but a PROPOSITION. To establish it we
-- have to give a PROOF.*

```
def Evens : PS Nat := fun n => n % 2 = 0
```

-- '4 ∈ Evens' unfolds to '4 % 2 = 0', so a computation proves it.

```
example : 4 ∈ Evens := show 4 % 2 = 0 by decide
```

```
#guard_msgs(drop error) in
```

```
example : 5 ∈ Evens := show 5 % 2 = 0 by decide -- fails: 5 % 2 = 1
```

*-- 'n ∈ Evens' IS 'n % 2 = 0', up to unfolding: a proof of one is a
-- proof of the other, with nothing to convert.*

```
example (n : Nat) (h : n % 2 = 0) : n ∈ Evens := h
```

Every quantifier ranges over a type

V is not a type, so $\forall x : V$ is not well formed. Quantify over S and guard:

$\forall x \in V. P x$ becomes

$\forall x : S. (x \in V \Rightarrow P x)$

$S = \mathbb{N}$, $V =$ the even numbers, $P x = "x \cdot x \text{ is even}"$



even x : the guard holds, $P x$ must be proved

odd x : the guard fails, nothing to prove

With $\wedge$ instead: every x in S would have to be in V . Too strong.

$\exists x \in V. P x$ becomes

$\exists x : S. (x \in V \wedge P x)$

$S = \mathbb{N}$, $V =$ the even numbers, $P x = "x > 3"$



one x in V that has P : $x = 4$

$(4 \in V \text{ and } 4 > 3)$

With $\Rightarrow$ instead: $x = 1$ would do, since $1 \notin V$ makes the $\Rightarrow$ vacuous. Says nothing.

Guarded quantifiers

```
--  $\forall x \in \text{Evens}. x \cdot x$  is even: quantify over Nat, guard with  $\Rightarrow$ .
example :  $\forall x : \text{Nat}, x \in \text{Evens} \rightarrow (x * x) \% 2 = 0 := \text{by}$ 
  intro x (hx :  $x \% 2 = 0$ ) -- any x, assuming the guard
  simp [Nat.mul_mod, hx]

--  $\exists x \in \text{Evens}. x > 3$ : guard with  $\wedge$ ; the witness 4 meets both halves.
example :  $\exists x : \text{Nat}, x \in \text{Evens} \wedge x > 3 :=$ 
  {4, show  $4 \% 2 = 0$  by decide, by decide}

-- With  $\Rightarrow$  instead,  $x = 1$  would do:  $1 \notin \text{Evens}$  makes the  $\Rightarrow$  vacuous.
example :  $\exists x : \text{Nat}, x \in \text{Evens} \rightarrow x > 1000 :=$ 
  {1, fun h => absurd (show  $1 \% 2 = 0$  from h) (by decide)}
```

A proof about sets

With $V \setminus W := \lambda x:S. (x \in V \wedge \neg(x \in W))$ and $W^c := \lambda x:S. \neg(x \in W)$: if $V \subseteq W^c$, then $V \setminus W = V$.

- (a) $S : *_s \mid V, W : ps(S)$
- (b) $x : S \mid v : x \in V \setminus W$
- (2) $a_2 := \dots$ use $\wedge$ -el₁ on $v \dots : x \in V$ $v : x \in V \wedge \neg(x \in W)$, by (conv)
- (4) $a_4(S, V, W) := \dots$ use $\Rightarrow$ -in and $\forall$ -in $\dots : V \setminus W \subseteq V$
- (c) $u : V \subseteq W^c$
- (d) $x : S \mid v : x \in V$
- (5) $a_5 := u x v : x \in W^c$ $\forall$ -el, $\Rightarrow$ -el
- (8) $a_8 := \dots$ use $\wedge$ -in on v and $a_5 \dots : x \in V \setminus W$ $x \in W^c =_\beta \neg(x \in W)$
- (10) $a_{10}(S, V, W, u) := \dots : V \subseteq V \setminus W$
- (12) $a_{12}(S, V, W) := \dots$ use $\wedge$ -in on a_4 and a_{10} , then $\Rightarrow$ -in $\dots : (V \subseteq W^c) \Rightarrow (V \setminus W = V)$

Each “use ...” names one rule. Only a_{10} depends on u .

N&G §13.2, Fig. 13.4

The same proof, as a script

```
-- The flag derivation, as a script. 'intro x v' raises the flags x : S
-- and v; the hints "use  $\wedge$ -el1", "use  $\wedge$ -in", "use  $\Rightarrow$ -in and  $\forall$ -in" become
-- '.1', '{_, _}' and 'intro'.
theorem diff_eq {S : Type} (V W : PS S) (u : V  $\sqsubseteq$  compl W) :
  IS (diff V W) V := by
  constructor
  · intro x v -- lines (1)-(4): no use of u
    exact v.1 -- (2)  $\wedge$ -el1 on v : x  $\in$  V  $\wedge$   $\neg$ (x  $\in$  W)
  · intro x v -- lines (5)-(10)
    have a5 : x  $\in$  compl W := u x v -- (5) u x v : x  $\in$  Wc
    exact {v, a5} -- (7)-(8)  $\wedge$ -in: x  $\in$  V \ W

-- The second half needs u and the first does not, and the parameter
-- lists show it (a4(S,V,W) against a10(S,V,W,u)).
```

A specification is a predicate, that is, a subset

a specification, in words

the sorted lists

the inputs on which f returns 0

the states from which a loop terminates

an assertion about a program state

“ P entails Q ”, P the stronger

a predicate transformer

as a predicate

$\lambda xs:List\mathbb{N}. sorted(xs) : ps(List\mathbb{N})$

$\lambda n:\mathbb{N}. (f\ n =_S 0) : ps(\mathbb{N})$

$\lambda s:State. \dots : ps(State)$

$\lambda s:State. (0 < s) : ps(State)$

$P \subseteq Q$

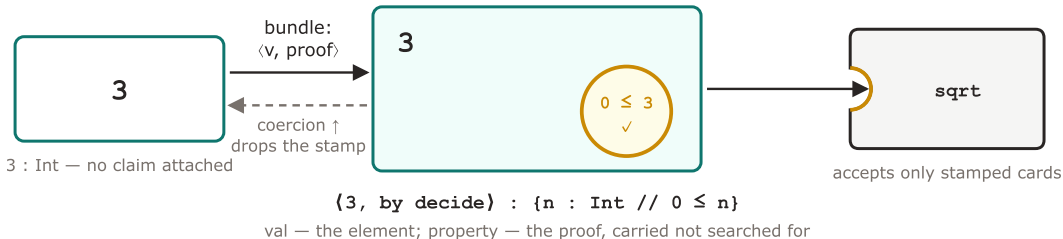
a map $ps(State) \rightarrow ps(State)$

“The program satisfies it” is a membership $x \in V$, to be proved. Nothing is added to the type theory to state a specification.

N&G §13.2

The other route: carry the proof

A subtype value is the element with its proof stamped on.



subsets-as-TYPES were weighed and rejected:
checking $M : \{x \mid x \geq 0\}$ means FINDING a proof — undecidable.
A subtype keeps typing decidable: the proof is an explicit component, checked, never searched.

The proof travels with the element: checked, never searched for.

Subtypes in Lean

```
-- Bundle the element WITH its proof:
-- the proof is carried, not searched for.
def NonNeg := {n : Int // 0 ≤ n}

def three : NonNeg := ⟨3, by decide⟩
#eval three.val      -- 3
#check three.property
-- three.property : 0 ≤ three.val

-- The cost: '3 : Int' and 'three : NonNeg'
-- are different terms of different types;
-- a coercion moves between them.
-- 'Fin n' is '{i : Nat // i < n}'.
```

Extensionality is an axiom

Two subsets with the same elements need not be equal: `=` compares the terms, and no rule looks inside a function. So it is added as an axiom.

-- Same elements, different terms: $0 + n$ does not compute to n .

```
def A : PS Nat := fun n => 0 + n = 5
```

```
def B : PS Nat := fun n => n = 5
```

```
theorem same_elements :  $\forall x, x \in A \leftrightarrow x \in B$  := by
```

```
  intro x; show 0 + x = 5  $\leftrightarrow$  x = 5; rw [Nat.zero_add]
```

```
#guard_msgs(drop error) in
```

```
example : A = B := rfl -- fails: equality compares the terms
```

-- The axioms: functions equal pointwise, propositions equal if $\leftrightarrow$.

```
theorem setext { $\alpha$  : Type} {V W : PS  $\alpha$ } (h :  $\forall x, x \in V \leftrightarrow x \in W$ ) : V = W :=
```

```
  funext (fun x => propext (h x))
```

```
example : A = B := setext same_elements
```

```
#print axioms setext -- [propext, Quot.sound]: funext comes from quotients
```

N&G §13.2, Fig. 13.5

What is still missing: numbers

Equality, orders and sets are definitions. Numbers are not: in λD they are six primitive definitions, and they do not compute.

The Peano axioms in λD

-
- (1)–(3) $\mathbb{N} := \perp\!\!\!\perp : *_S, \quad 0 := \perp\!\!\!\perp : \mathbb{N}, \quad s := \perp\!\!\!\perp : \mathbb{N} \rightarrow \mathbb{N}$ the set, zero, successor
- (4) $ax-nat_1 := \perp\!\!\!\perp : \forall x:\mathbb{N}. \neg(sx =_{\mathbb{N}} 0)$
- (5) $ax-nat_2 := \perp\!\!\!\perp : \forall x, y:\mathbb{N}. (sx =_{\mathbb{N}} sy \Rightarrow x =_{\mathbb{N}} y)$
- (6) $ax-nat_3(P) := \perp\!\!\!\perp : (P0 \wedge \forall x:\mathbb{N}. (Px \Rightarrow P(sx))) \Rightarrow \forall x:\mathbb{N}. Px$ induction
-

And without recursive definitions, *fac* is not a program: it cannot be executed, so *fac*(3) = 6 requires a proof.

■ Next lecture

Inductive types: two constructors in place of six axioms, and *fac* 3 = 6 by rfl.

N&G §8.2, §14.1

References I

Avigad, J., L. de Moura, S. Kong, and S. Ullrich (2026). *Theorem Proving in Lean 4*. With contributions from the Lean community; tested against Lean 4.33.0. URL: https://lean-lang.org/theorem_proving_in_lean4/ (visited on 08/20/2026).

Frege, G. (1893). *Grundgesetze der Arithmetik*. Jena: Pohle.

Nederpelt, R. and H. Geuvers (2014a). “Mathematics in λD : a first attempt”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 12, pp. 257–278.

— (2014b). “Numbers and arithmetic in λD ”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 14, pp. 305–348.

— (2014c). “Sets and subsets”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 13, pp. 279–304.

— (2014d). *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press.

References II

N&G chapters 12 and 13 throughout. Ch. 12: §12.1 (Lemma 12.1.2, Fig. 12.1), §12.2 (Figs. 12.2–12.4, Rem. 12.2.1–12.2.2), §12.3 (Figs. 12.5–12.6), §12.4 (Fig. 12.7, Rem. 12.4.1, Figs. 12.8–12.9), §12.5 (Figs. 12.10–12.12), §12.6 (Figs. 12.13–12.15, p. 271), §12.7 (Figs. 12.16–12.18, Lemma 12.7.1, Rem. 12.7.2). Ch. 13: §13.1, §13.2 (Figs. 13.1–13.5, Ex. 13.2.1, Rem. 13.2.2), §13.3 (Fig. 13.6). §14.13 for irrelevance of proof; §10.5 for the confluence and normalisation results quoted on the rewriting slide. The Lean slides follow *Theorem Proving in Lean 4*: §4.1–4.3 (equality, `calc`), §5.6–5.7 (`rw`, `simp`), ch. 11 (`conv`) and §12.2–12.5 (`propext`, `funext`, `choice`).