



Computational Logic in the Era of AI

Definitions and Environments

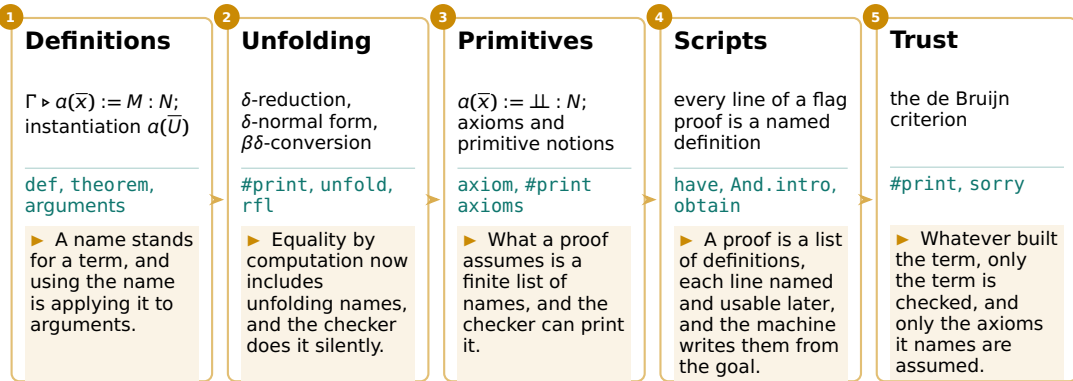
Sergey Mechtaev
mechtaev@pku.edu.cn

Term 1 • 26–27

This lecture

↓ you are here

■ theory ■ in Lean



History: every mathematics book is an environment



Euclid

fl. 300 BC

Definitions,
postulates, and
propositions that
cite earlier ones by
number.

λD makes Euclid's structure a formal object: an environment of definitions, some with a body and some without. Photos: Wikimedia.

c. 300 BC Euclid's *Elements*: definitions with a body, postulates without one, and every proposition built from what came before.

1968 De Bruijn's Automath: a "book" is a list of lines, each a definition or a primitive notion, and a machine checks the book.

1990s Barendregt names the de Bruijn criterion: whatever builds the term, a small kernel checks it.

2021 Lean 4: `#print axioms` lists the primitive notions a theorem rests on.



N. G. de Bruijn

1918–2012

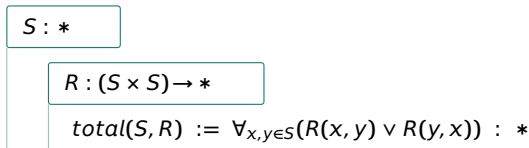
Definitions as
first-class citizens
of a type theory.

What a definition is

I Definition (definiendum, definiens)

In $a := \mathcal{E}$, the name a is the **definiendum**, the defined constant; the expression $\mathcal{E}$ is the **definiens**.

A definition needs a setting: “total” needs a set and a relation on it. The setting is a **context**, written in flags.



The list (S, R) is the **parameter list**: the subjects of the context flags, in order.
N&G Def. 8.3.1, Rem. 8.3.2, Fig. 8.1

A series of definitions in one context

- (a) $S : *$
- (b) $R : (S \times S) \rightarrow *$
- (1) $\text{reflexive}(S, R) := \forall_{x \in S} (R(x, x)) : *$
- (2) $\text{antisymmetric}(S, R) := \forall_{x, y \in S} ((R(x, y) \wedge R(y, x)) \Rightarrow x = y) : *$
- (3) $\text{transitive}(S, R) := \forall_{x, y, z \in S} ((R(x, y) \wedge R(y, z)) \Rightarrow R(x, z)) : *$
- (4) $\text{partially-ordered}(S, R) :=$
 $\text{reflexive}(S, R) \wedge (\text{antisymmetric}(S, R) \wedge \text{transitive}(S, R)) : *$

Left of $:=$, (S, R) is a **parameter list**; in line (4) the same lists are **instantiations**. Hence the order: (4) after (1) to (3).

N&G §8.5, Fig. 8.2

The series, in Lean

```
-- The two flags are the two binders; the type follows ':', the definiens ':='.
def Reflexive (S : Type) (R : S → S → Prop) : Prop := ∀ x, R x x
def Antisymm (S : Type) (R : S → S → Prop) : Prop :=
  ∀ x y, R x y → R y x → x = y
def Transitive (S : Type) (R : S → S → Prop) : Prop :=
  ∀ x y z, R x y → R y z → R x z
def PartialOrder (S : Type) (R : S → S → Prop) : Prop :=
  Reflexive S R ∧ Antisymm S R ∧ Transitive S R -- (4) after (1)-(3)

#check @Reflexive -- @Reflexive : (S : Type) → (S → S → Prop) → Prop
```

Outside its definition the constant takes the flags as arguments: #check shows the parameter list as a Π -type.

Using a definition: instantiation

A constant is born in its definition, then lives by **instantiation** of its parameters.

	first flag	second flag
	$S : *$	$R : (S \times S) \rightarrow *$
instantiation	$\mathbb{R} : *$	$\leq : (\mathbb{R} \times \mathbb{R}) \rightarrow *$
instantiation	$\mathbb{N}^+ : *$	$: (\mathbb{N}^+ \times \mathbb{N}^+) \rightarrow *$

Instantiating S changes the type of R : in the first row $\leq$ is checked against $((S \times S) \rightarrow *)[S := \mathbb{R}]$, the flag's type after the earlier substitution.

$total(\mathbb{N}^+, |)$ is a correct instantiation, and false.

N&G §8.4, Rem. 8.4.1

Instantiation is application

```
-- Two instantiations; the type of R is computed from the choice of S.
#check Reflexive Nat ( $\cdot \leq \cdot$ )          -- Reflexive Nat (fun x y => x ≤ y) : Prop
#check Reflexive String ( $\cdot \neq \cdot$ )      -- Reflexive String (fun x y => x ≠ y) : Prop

-- Under S := Nat, a relation on strings is refused:
#guard_msgs(drop error) in
#check Reflexive Nat (fun (s t : String) => s = t)
-- Application type mismatch: the argument has type String → String → Prop
-- but is expected to have type Nat → Nat → Prop

-- A correct instantiation, a false proposition, and so refutable:
example : Reflexive Nat ( $\cdot < \cdot$ ) → False := fun h => Nat.lt_irrefl 0 (h 0)
```

Flags in Lean: variable and section

```
-- 'variable' raises the flag; 'section ... end' is its pole.
section
variable (S : Type) (R : S → S → Prop)

def Refl  : Prop := ∀ x, R x x                -- parameters unwritten
def Trans : Prop := ∀ x y z, R x y → R y z → R x z
def Preord : Prop := Refl S R ∧ Trans S R      -- and written when used
end                                              -- the flag is hauled down

#check @Preord      -- Preord : (S : Type) → (S → S → Prop) → Prop
```

Four kinds of definition

Write $*_s$ for $*$ read as “sets” and $*_p$ for $*$ read as “propositions”; the subscript is a reading aid, not a new sort. $\Gamma \triangleright a(\dots) := \mathcal{E} : T$ writes the flags Γ on one line, before the definition they govern.

(sets)	$\Gamma_1 \triangleright$	$A(\dots) := \mathcal{E}_1$	$: *_s$
(objects)	$\Gamma_2 \triangleright$	$a(\dots) := \mathcal{E}_2$	$: S \text{ with } S : *_s$
(propositions)	$\Gamma_3 \triangleright$	$B(\dots) := \mathcal{E}_3$	$: *_p$
(proofs)	$\Gamma_4 \triangleright$	$b(\dots) := \mathcal{E}_4$	$: P \text{ with } P : *_p$

The fourth names a proof: under PAT, a term like any other. It is what makes a theorem applicable.

The one asymmetry: a notion is defined in order to be unfolded; a proof is used through its type, and its body is not unfolded. N&G Not. 8.7.1, Figs. 8.4–8.5, §16.1

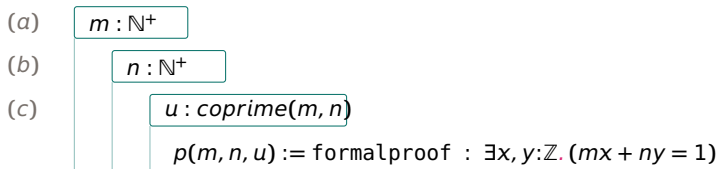
A theorem is a definition

```
-- 'theorem' for proofs, 'def' for the other three kinds.
def      Nat2    : Type := Nat           -- sets
def      two     : Nat  := 2             -- objects
def      Even    (n : Nat) : Prop := ∃ k, n = 2*k -- propositions
theorem  even4    : Even 4 := ⟨2, rfl⟩    -- proofs

#print two           -- def ...two : Nat :=
                    -- 2
#print even4         -- theorem ...even4 : Even 4 :=
                    -- Exists.intro 2 rfl
```

Giving names to proofs

Bézout's Lemma: if m, n are coprime, $\gcd(m, n) = 1$, there are integers x, y with $mx + ny = 1$. The third flag is an assumption: a proof, declared like any other variable.



Without the name, using the lemma at 55, 28 means copying the proof with 55 and 28 substituted throughout. With it, $p(55, 28, U)$ is the specialised proof, for any $U : \text{coprime}(55, 28)$.

A theorem is applied by instantiation.

N&G §8.6–8.8, Figs. 8.3, 8.6–8.7

A theorem applied by instantiation, in Lean

-- 1. A theorem with three data parameters and one proof parameter, u .
`theorem le_add_right_of_le (a b k : Nat) (u : a ≤ b) : a + k ≤ b + k :=
 Nat.add_le_add_right u k`

-- 2. Instantiate the data: the type of u becomes $3 \leq 5$.
`#check le_add_right_of_le 3 5 10` -- $3 \leq 5 \rightarrow 3 + 10 \leq 5 + 10$

-- 3. Supply $U : 3 \leq 5$ as the last argument: the specialised theorem.
`example : 3 + 10 ≤ 5 + 10 := le_add_right_of_le 3 5 10 (by decide)`

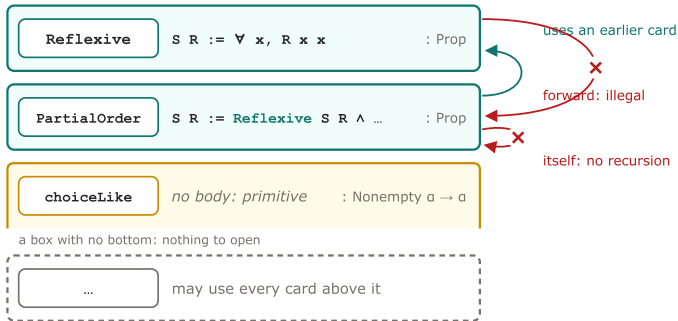
This is $p(55, 28, U)$: instantiate the data parameters, then supply the proof parameter.

The environment Δ

Definition (environment)

An **environment** Δ is a finite list of definitions; an **extended judgement** has the form $\Delta; \Gamma \vdash M : N$. Δ binds the **constants** in $M : N$, Γ the free variables.

Δ : written downwards, read upwards



Once added, a definition is never erased: Δ only grows.

N&G Def. 9.1.2, 9.2.1, Rem. 9.2.2, 9.2.4

Two rules: adding a definition, and using one

■ (*def*) — adding a definition

$$\frac{\Delta; \Gamma \vdash K : L \quad \Delta; \bar{x}:\bar{A} \vdash M : N}{\Delta, \mathcal{D}; \Gamma \vdash K : L} \quad (\alpha \text{ fresh in } \Delta) \quad \text{with } \mathcal{D} \equiv \bar{x}:\bar{A} \triangleright \alpha(\bar{x}) := M : N$$

$\bar{x}:\bar{A}$ are the parameter flags $x_1:A_1, \dots, x_n:A_n$; $\bar{U}$ below are the arguments supplied for them.

■ (*inst*) — using one

$$\frac{\Delta; \Gamma \vdash * : \square \quad \Delta; \Gamma \vdash \bar{U} : \overline{A[\bar{x} := \bar{U}]}}{\Delta; \Gamma \vdash \alpha(\bar{U}) : N[\bar{x} := \bar{U}]} \quad ()$$

The first premiss says only that Δ and Γ are well-formed; it also covers an empty parameter list.

Instance. (*def*) adds *reflexive* to Δ from $\Delta; S:*, R:(S \times S) \rightarrow * \vdash \forall_{x \in S} (R(x, x)) : *$; then (*inst*) gives *reflexive*($\mathbb{N}, \leq$) : $*$.

Not an instance. *reflexive*($\mathbb{N}, =_{\mathbb{R}}$): the second premiss asks for $U_2 : (\mathbb{N} \times \mathbb{N}) \rightarrow *$, and $=_{\mathbb{R}}$ has type $(\mathbb{R} \times \mathbb{R}) \rightarrow *$.
N&G Def. 9.3.1, 9.4.7, §9.4

A file is an environment

```
-- A file is  $\Delta$ ; 'variable' and a declaration's binders are  $\Gamma$ .
variable (S : Type)
def Reflexive (R : S → S → Prop) : Prop := ∀ x, R x x
def PartialOrder (R : S → S → Prop) : Prop := Reflexive S R

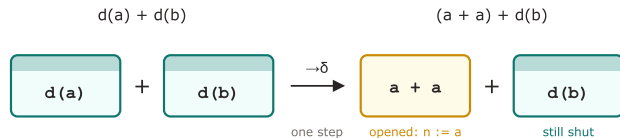
#print PartialOrder
-- def ...PartialOrder : (S : Type) → (S → S → Prop) → Prop :=
-- fun S R => Reflexive S R
--  $\Gamma$  has become  $\lambda$ .
```

δ -reduction unfolds one occurrence

Definition (one-step δ -reduction)

If $\Gamma \triangleright a(\bar{x}) := M : N$ is in Δ , then $a(\bar{U}) \rightarrow_{\delta} M[\bar{x} := \bar{U}]$, also inside any larger expression.

With $n:\mathbb{N} \triangleright d(n) := n + n : \mathbb{N}$ in Δ :



β is different: $(\lambda n. n + n) a \rightarrow_{\beta} a + a$ replaces every bound n at once.

One occurrence at a time; the other d stays folded.

N&G Def. 9.5.1, Rem. 9.5.3

rw takes one δ -step

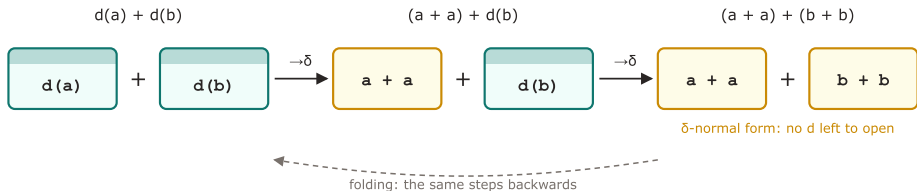
```
def d (n : Nat) : Nat := n + n          -- n:ℕ ▷ d(n) := n + n : ℕ, now in Δ

-- 'rw [d]' unfolds one occurrence of d: the first it finds, here d a.
example (a b : Nat) : d a + d b = (a + a) + (b + b) := by
  rw [d]      -- ⊢ a + a + d b = a + a + (b + b)      d b is still folded
  rw [d]      -- ⊢ a + a + (b + b) = a + a + (b + b)  both sides agree: closed
```

δ -normal form: nothing left to unfold

Definition (δ -reduction, δ -conversion, δ -normal form)

$M \rightarrow_{\delta} N$: zero or more δ -steps. $M =_{\delta} N$: steps in either direction, a backward step being a **folding**. K is in **δ -normal form** when no constant with a body occurs in it.



N&G Def. 9.5.4, 9.5.5, 9.5.7

unfold takes every δ -step for one constant

-- 'unfold d' unfolds every occurrence of d: the δ -normal form with respect to d.

```
example (a b : Nat) : d a + d b = (a + a) + (b + b) := by
```

```
  unfold d    --  $\vdash a + a + (b + b) = a + a + (b + b)$ 
```

```
  rfl
```

-- With respect to d only: a constant defined through d stays folded.

```
def e (n : Nat) : Nat := d n + 1
```

```
example (a : Nat) : e a = (a + a) + 1 := by
```

```
  unfold e    --  $\vdash d\ a + 1 = a + a + 1$ 
```

d is still folded

```
  unfold d    --  $\vdash a + a + 1 = a + a + 1$ 
```

```
  rfl
```

$\beta\delta$ -conversion

Definition ($\beta\delta$ -conversion, and (*conv*) again)

$M =_{\beta\delta} N$ when the two are linked by a chain of steps, each a β -step or a δ -step, in either direction.

$$\frac{\Delta; \Gamma \vdash A : B \quad \Delta; \Gamma \vdash B' : s}{\Delta; \Gamma \vdash A : B'} \text{ (conv) if } B =_{\beta\delta} B'$$

relation	confluent	terminating	normal form
δ alone	yes	yes	unique; only primitive constants left
β alone	yes	yes	unique
$\beta\delta$ combined	yes	yes	unique

Termination of the combination is a theorem of its own, and it is why *rfl* terminates.

N&G Def. 9.7.2, 9.7.3, Thm. 6.3.11, 6.3.14, 10.5.2–10.5.8

rfl does β and δ

```
-- 'rfl' decides  $\beta\delta$ -convertibility: it normalises both sides and compares.
def d (n : Nat) : Nat := n + n

example : d 3 = 6 := rfl          -- one  $\delta$ -step, then arithmetic
example : d = fun n => n + n := rfl

-- Three definitions, the third using the other two; unfolding all of them:
def a (x y : Int) : Int := x*x + y*y
def b (x y : Int) : Int := 2 * (x * y)
def c (x y : Int) : Int := a x y + b x y
def lemma' (x y : Int) : Prop := c x y = (x + y) * (x + y)

example (u v : Int) : lemma' u v = (u*u + v*v + 2*(u*v) = (u+v)*(u+v)) := rfl
-- Three  $\delta$ -steps, in whichever order: the normal form is unique.
```

Unfolding is a policy

```
-- Which constants the checker unfolds on its own is set per definition.
abbrev Fast : Type := Nat      -- @[reducible]: always unfolded
def      Norm : Type := Nat    -- default: unfolded when needed
@[irreducible] def Slow : Type := Nat  -- never unfolded automatically

example : Fast = Nat := rfl
example : Norm = Nat := rfl
#guard_msgs(drop error) in
example : Slow = Nat := rfl      -- Type mismatch: rfl has type ?m = ?m
                                -- but is expected to have type Slow = Nat
                                -- the policy, lifted for one declaration
unseal Slow in                  -- accepted:  $\delta$  is available again
example : Slow = Nat := rfl
```

δ -reduction is one relation on paper; a checker decides convertibility with as few unfoldings as it can, and each definition says how eagerly it may be opened.

Descriptive and primitive definitions

Definition

$\Gamma \triangleright a(\bar{x}) := K : L$ is a **descriptive** definition; a is a proper constant.

$\Gamma \triangleright a(\bar{x}) := \perp\!\!\!\perp : L$ is a **primitive** definition; a is a primitive constant.

$\perp\!\!\!\perp$ marks the absence of a definiens: a primitive constant cannot be unfolded.
Axioms and primitive notions enter this way:

$\emptyset \triangleright \mathbb{N} := \perp\!\!\!\perp : *_s$	a set, postulated
$\emptyset \triangleright 0 := \perp\!\!\!\perp : \mathbb{N}, \quad s := \perp\!\!\!\perp : \mathbb{N} \rightarrow \mathbb{N}$	objects, postulated
$P : \mathbb{N} \rightarrow *_p \triangleright \text{ind}(P) := \perp\!\!\!\perp : P 0 \Rightarrow (\forall_n (P n \Rightarrow P(s n))) \Rightarrow \forall_n (P n)$	an axiom scheme
$\emptyset \triangleright i_{ET} := \perp\!\!\!\perp : \prod \alpha : * . (\alpha \vee \neg \alpha)$	the excluded third

(def-prim) and *(inst-prim)* are *(def)* and *(inst)* with the premiss about M
replaced by $\Gamma \vdash N : s$.

N&G Def. 10.2.1, 10.4.2, Ex. 10.2.2, Fig. 10.1

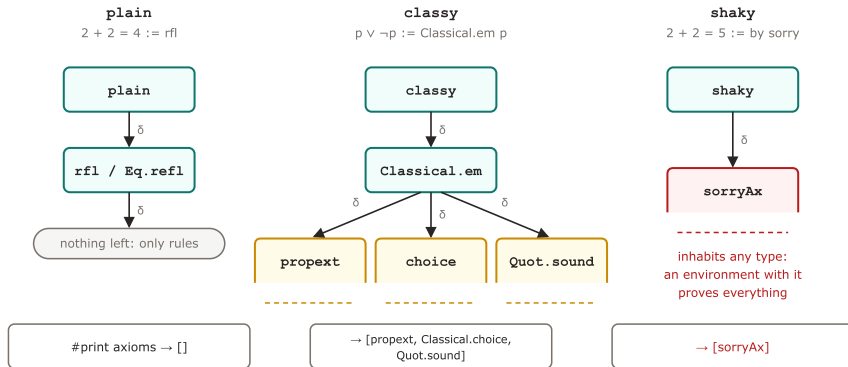
axiom is a definition with no body

```
-- A primitive definition has no body: 'axiom', and no ':='.
axiom choiceLike ( $\alpha$  : Type) : Nonempty  $\alpha$   $\rightarrow$   $\alpha$ 

#print choiceLike
-- axiom choiceLike : ( $\alpha$  : Type)  $\rightarrow$  Nonempty  $\alpha$   $\rightarrow$   $\alpha$ 
-- no ':=': nothing to unfold

-- Lean's own three primitives are of exactly this kind:
#check @propext
#check @Classical.choice
#check @Quot.sound
```

What survives δ -normalisation



Unfold every proper constant; what survives is exactly the primitive ones.

`#print axioms` opens every box that has a bottom and writes down the ones that do not; `sorry` is a bottomless box that fits any type.

The audit

```
-- '#print axioms': the primitive constants left after  $\delta$ -normalisation.
theorem plain : 2 + 2 = 4 := rfl
theorem classy (p : Prop) : p  $\vee$   $\neg$ p := Classical.em p

#print axioms plain      -- 'plain' does not depend on any axioms
#print axioms classy
-- 'classy' depends on axioms: [propext, Classical.choice, Quot.sound]

-- One caveat. A recursor has no body either, yet:
#print axioms Nat.rec    -- 'Nat.rec' does not depend on any axioms
-- The audit lists the constants declared as axioms; the ones every
-- 'inductive' adds, its recursor, it trusts silently.
```

Why three names, and not one

```
-- On paper classical logic adds one primitive; the audit reported three.
#print axioms Classical.choice -- depends on axioms: [Classical.choice]
#print axioms propext         -- depends on axioms: [propext]
#print axioms Quot.sound      -- depends on axioms: [Quot.sound]

-- These two are not: both are theorems, and the audit says what they cost.
#print axioms funext          -- depends on axioms: [Quot.sound]
#print axioms Classical.em
-- 'Classical.em' depends on axioms: [propext, Classical.choice, Quot.sound]

-- 'funext' is proved from quotients; 'Classical.em' from choice, propext
-- and funext, Diaconescu's theorem. The excluded third is derived.
```

Diaconescu (1975): choice implies the excluded middle, so Lean derives $\alpha \vee \neg \alpha$ from its three primitives. N&G §7.4, Fig. 11.16; TPIL §12.2–12.6

sorry compiles

```
-- 'sorry' is a primitive constant that inhabits any type: an environment
-- containing it proves everything.
theorem shaky : 2 + 2 = 5 := by sorry

#print axioms shaky      -- 'shaky' depends on axioms: [sorryAx]
#check @sorryAx          -- sorryAx : ( $\alpha$  : Sort u_1)  $\rightarrow$  Bool  $\rightarrow$   $\alpha$ 

-- It compiles, with a warning. Only the audit reveals it.
```

An axiom is a promise

```
-- A primitive definition passes every check whether or not it makes sense.  
--  $k(A,B) := \perp\perp : (A \Rightarrow B) \Rightarrow A$  looks like Peirce's law, and derives False:  
axiom bad (A B : Prop) : (A  $\rightarrow$  B)  $\rightarrow$  A  
  
theorem everything : False := bad False False (fun h => h)  
  
#print axioms everything    -- 'everything' depends on axioms: [bad]  
  
-- The kernel is not fooled: it reports the culprit by name. An axiom is  
-- a promise, and '#print axioms' is where the promises are collected.
```

Logic in λD : the hidden definitions, written down

Last lecture $\perp$, $\neg$ and $\wedge$ were abbreviations; in λD they are descriptive definitions:

$$(\mathcal{D}_1) \quad \perp() := \Pi\alpha: * . \alpha : *$$

$$A : *$$

$$(\mathcal{D}_2) \quad \neg(A) := A \rightarrow \perp() : *$$

$$A, B : *$$

$$(\mathcal{D}_3) \quad \wedge(A, B) := \Pi C: * . (A \rightarrow B \rightarrow C) \rightarrow C : *$$

Each line stands for a full derivation, ten lines for $\mathcal{D}_2$ alone; the definition records that its statement is derivable.

■ The convention: every statement becomes a definition

In a flag derivation, every statement $\Gamma \vdash M : N$ gets a fresh name:

$$\Gamma \triangleright c(\overline{x}) := M : N.$$

N&G §11.1–11.3, Figs. 11.1–11.2

The natural-deduction rules, as defined constants

Each natural-deduction rule becomes a definition named after it: its parameter list is its context, its body the rule's derivation.

flags	definition
$A, B : *p \mid u : A \rightarrow B$	$\Rightarrow\text{-in}(A, B, u) := u : A \Rightarrow B$
$A : *p \mid u : \perp$	$\perp\text{-el}(A, u) := u A : A$
$A : *p \mid u : \neg A \mid v : A$	$\neg\text{-el}(A, u, v) := u v : \perp$
$A, B : *p \mid u : A \mid v : B$	$\wedge\text{-in}(A, B, u, v) := \lambda C : *p. \lambda w : A \Rightarrow B \Rightarrow C. w u v : A \wedge B$
$A, B : *p \mid u : A \wedge B$	$\wedge\text{-el}_1(A, B, u) := u A (\lambda v : A. \lambda w : B. v) : A$
$A, B : *p \mid u : A$	$\vee\text{-in}_1(A, B, u) := \lambda C : *p. \lambda v : A \Rightarrow C. \lambda w : B \Rightarrow C. v u : A \vee B$
$A, B, C : *p \mid u : A \vee B \mid v : A \Rightarrow C \mid w : B \Rightarrow C$	$\vee\text{-el}(A, B, C, u, v, w) := u C v w : C$

Two styles for one proof of $A \Rightarrow \neg\neg A$: naming every rule, $a_3(A) := \Rightarrow\text{-in}(A, \neg\neg A, \lambda u : A. a_2(A, u))$, or writing the term, $a_3(A) := \lambda u : A. \lambda v : \neg A. v u$. Shorter here; for $\wedge$ and $\vee$ the names win.

The named rules are the library

v-elimination defined once, as a named line

```
A, B, C : *p  
|  
| u : A → B | v : A → C | w : B → C  
|  
| v-elim(A, B, C, u, v, w) := u C v w : C
```

parameters that only repeat the context are suppressed —
Lean writes them in braces

```
#check @Or.elim
```

```
@Or.elim : ∀ {a b c : Prop}, a → b → (a → c) → (b → c) → c  
                                  u                  v                  w
```

Parameters that only repeat the context may be left out; {a b c : Prop} is that convention, and @ switches it off.

N&G §11.7, Fig. 11.11

The rules, by their library names

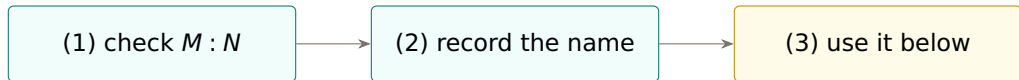
```
-- The library holds the named rules; the parameters that only repeat
-- the context, A B C, are suppressed by '{}'.
#check @And.intro      -- @And.intro :  $\forall \{a\ b : Prop\}, a \rightarrow b \rightarrow a \wedge b$ 
#check @Or.elim        -- @Or.elim :  $\forall \{a\ b\ c : Prop\}, a \vee b \rightarrow (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow c$ 
#check @absurd         -- @absurd :  $\{a : Prop\} \rightarrow \{b : Sort\ u\_1\} \rightarrow a \rightarrow \neg a \rightarrow b$ 

variable (A B C : Prop)
example (u : A) (v : B) : A  $\wedge$  B := And.intro u v      --  $\wedge$ -in(A,B,u,v)
example (u : A  $\wedge$  B) : A := u.left                    --  $\wedge$ -el1(A,B,u)
example (u : A) : A  $\vee$  B := Or.inl u                  --  $\vee$ -in1(A,B,u)
example (u : A  $\vee$  B) (v : A  $\rightarrow$  C) (w : B  $\rightarrow$  C) : C :=
  Or.elim u v w                                         --  $\vee$ -el(A,B,C,u,v,w)
```

One line, three meanings

A flag derivation writes each step as a line $c(\bar{x}) := M : N$. Such a line means three things at once.

1. the statement $M : N$ is derivable in the current environment and context;
2. the definition is appended to Δ , under the name c ;
3. every later line may use $c(\bar{x}) : N$.



Check, record, use. That is what have does, and it is why have is the most important tactic in the language.

N&G §11.3

have is a flag line

-- 'have' is a flag line: the statement is checked, recorded under a name, and usable below.

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  have h1 : p := hp           -- checked, recorded, usable below
  have h2 : q := hq
  exact And.intro h1 h2
```

The goal state after a have is the flag derivation after that line: one more name in scope, the goal unchanged.

show, suffices, refine

```
-- Three more ways to write a flag line.
-- 'show' restates the goal at a  $\beta\delta$ -convertible type: (conv), as a tactic.
example (p : Prop) (hp : p) :  $\neg\neg p$  := by
  show  $\neg p \rightarrow \text{False}$ 
  intro hnp; exact hnp hp

-- 'suffices' is 'have' read backwards: name what would be enough, use it
-- at once, and prove it afterwards.
example (p q : Prop) (hp : p) (h : p  $\rightarrow$  q) : q  $\wedge$  p := by
  suffices hq : q from ⟨hq, hp⟩
  exact h hp

-- 'refine' is 'exact' with holes: each '?_' becomes a goal, a flag still
-- to be hauled down.
example (p q : Prop) (hp : p) (hq : q) : (p  $\wedge$  q)  $\wedge$  p := by
  refine ⟨{?_, hq}, {?_}⟩
  · exact hp
  · exact hp
```

A derivation as a list of definitions

Last lecture's tautology $(A \vee B) \Rightarrow (\neg A \Rightarrow B)$, parameters suppressed:

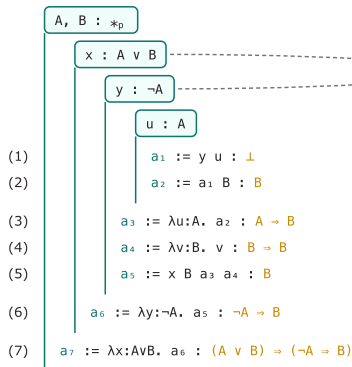
- | | | |
|-----|--|------------------------------------|
| (a) | $A, B : *_p$ | |
| (b) | $x : A \vee B$ | |
| (c) | $y : \neg A$ | |
| (d) | $u : A$ | |
| (1) | $a_1 := y u : \perp$ | $\neg\text{-el, on (c), (d)}$ |
| (2) | $a_2 := a_1 B : B$ | $\perp\text{-el, on (1)}$ |
| (3) | $a_3 := \lambda u:A. a_2 : A \Rightarrow B$ | $\Rightarrow\text{-in, on (2)}$ |
| (4) | $a_4 := \lambda v:B. v : B \Rightarrow B$ | $\Rightarrow\text{-in}$ |
| (5) | $a_5 := x B a_3 a_4 : B$ | $\vee\text{-el, on (b), (3), (4)}$ |
| (6) | $a_6 := \lambda y:\neg A. a_5 : \neg A \Rightarrow B$ | $\Rightarrow\text{-in, on (5)}$ |
| (7) | $a_7 := \lambda x:A \vee B. a_6 : (A \vee B) \Rightarrow (\neg A \Rightarrow B)$ | $\Rightarrow\text{-in, on (6)}$ |

Seven lines, seven names; unfolding (7) down to a_1 gives last lecture's term.

N&G §11.6–11.7

The script is the derivation, written from the bottom

the derivation



have = a named line; intro = a flag raised; exact/apply = the definien

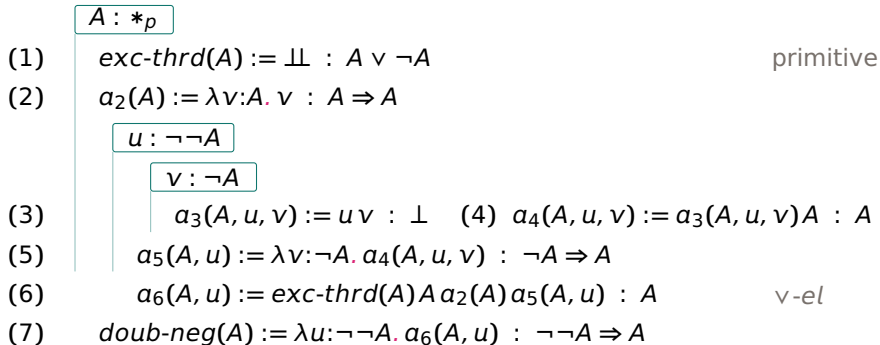
the Lean script

```
theorem t : (A ∨ B) → (¬A → B) := by
  intro x y
  have a3 : A → B := fun u => (y u).elim
  have a4 : B → B := fun v => v
  exact x.elim a3 a4
```

N&G Fig. 11.15; TPIL §5.4

Classical logic in λD

The excluded third is one primitive definition in the environment:



From *doub-neg* comes a strategy: to prove A , raise a flag $\neg A$ and derive $\perp$.

N&G §11.8, Rem. 11.8.1

The case split, and what the audit says of it

```
-- On paper  $\text{exc-thrd}(A) := \perp\perp : A \vee \neg A$  is primitive; in Lean it is a
-- theorem, and a script that uses it is classical:
theorem doubNeg (A : Prop) (u :  $\neg\neg A$ ) : A := by
  by_cases h : A                -- exc-thrd(A), as a case split
  · exact h
  · exact (u h).elim             --  $\neg\neg A$  and  $\neg A$  give  $\perp$ , and  $\perp$ -el

theorem doubNeg' (A : Prop) (u :  $\neg\neg A$ ) : A :=
  Classical.byContradiction u    -- assume  $\neg A$ , derive  $\perp$ 

theorem constructive (A : Prop) : A  $\rightarrow$   $\neg\neg A$  := fun a f => f a

#print axioms doubNeg           -- depends on axioms: [propext, Classical.choice, Quot.sound]
#print axioms constructive     -- 'constructive' does not depend on any axioms

by_cases is  $\vee$ -elimination on  $\text{exc-thrd}(A)$ ; Classical.byContradiction is line (7) by
name. Both are classical, and #print axioms says so; the constructive direction reports
nothing.
```

$\exists$ -elimination raises two flags

To use $\exists x:S. Px$ while proving A , one cannot extract a witness; the rule wants $\forall x:S. (Px \Rightarrow A)$, built by raising two flags:

$$\begin{array}{l} a(\dots) := \dots : \exists x:S. Px \\ \quad \boxed{x : S} \\ \quad \quad \boxed{u : Px} \\ \quad \quad \quad b(\dots, x, u) := \dots : A \\ \quad \quad c(\dots, x) := \lambda u:Px. b(\dots) : Px \Rightarrow A \\ \quad d(\dots) := \lambda x:S. c(\dots) : \forall x:S. (Px \Rightarrow A) \\ e(\dots) := \exists\text{-el}(S, P, a, A, d) : A \end{array}$$

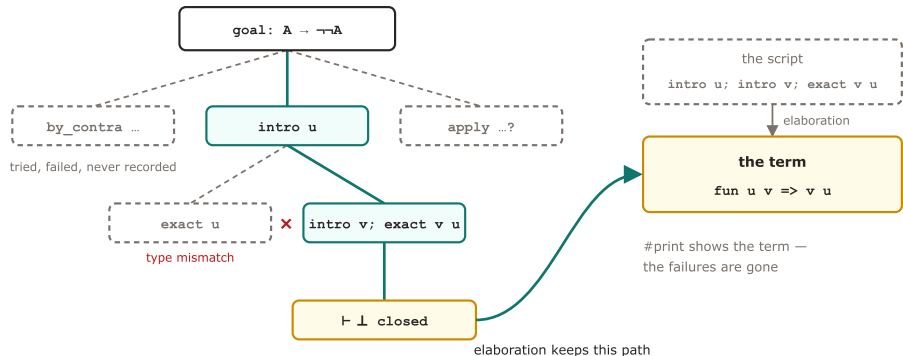
The goal A is unchanged; only the context is larger. That is obtain. Side condition: x not free in A .
N&G §11.10, Rem. 11.10.2

obtain is $\exists$ -elimination

```
-- To use  $\exists x:S. P x$  while proving  $A$ , raise two flags,  $x : S$  and  $u : P x$ ,
-- prove  $A$  under them, and  $\exists$ -el discharges both. That is 'obtain'.
example (S : Type) (P : S → Prop) (Q : Prop)
  (h :  $\exists x, P x$ ) (k :  $\forall x, P x \rightarrow Q$ ) : Q := by
  obtain ⟨x, hx⟩ := h      -- raises the flags  $x : S$  and  $hx : P x$ 
  exact k x hx            -- prove  $Q$  under them

--  $\exists x.Px \Rightarrow \forall y.(Py \Rightarrow Qy) \Rightarrow \exists z.Qz$ , a six-line derivation in two:
example (S : Type) (P Q : S → Prop) (u :  $\exists x, P x$ ) (v :  $\forall y, P y \rightarrow Q y$ ) :
   $\exists z, Q z$  := by
  obtain ⟨x, w⟩ := u      -- the flags  $x : S$ ,  $w : P x$ 
  exact ⟨x, v x w⟩        -- (1)  $v x$ , (2)  $a_1 w$ , (3)  $\exists$ -in, (5)  $\exists$ -el
-- The side condition:  $x$  must not occur free in the goal. Lean enforces
-- it: the goal may not mention 'x'.
```

Devised backwards, recorded forwards



A tactic script is the record of a search whose successful branch became a term; the failed attempts are not recorded.

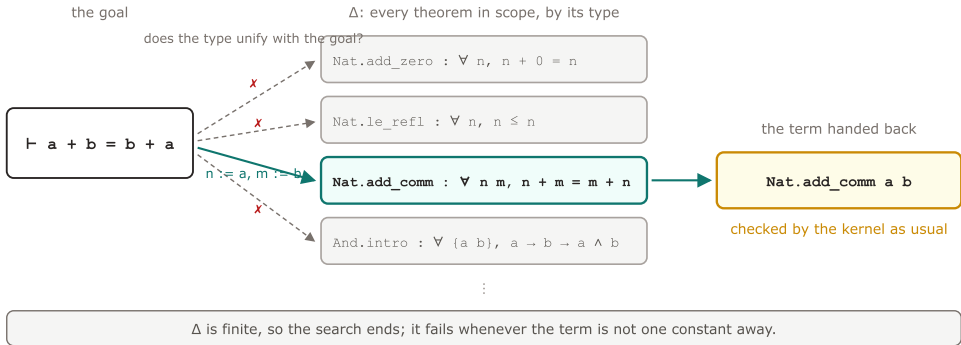
N&G Rem. 11.9.3

The script is not the term

```
-- A derivation is checked from top to bottom, but devised the other way:
-- from the goal, with the failed attempts left unrecorded.
theorem demorgan (A B : Prop) (h :  $\neg A \vee \neg B$ ) :  $\neg(A \wedge B)$  := by
  intro hab
  cases h with
  | inl ha => exact ha hab.left
  | inr hb => exact hb hab.right

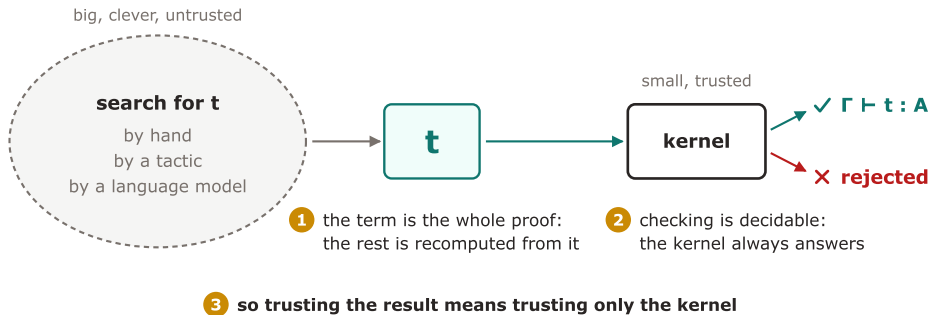
#print demorgan -- forwards: only the result survives
-- fun A B h hab =>
--   Or.casesOn (motive := fun t => h = t → False) h
--   (fun ha h => ha hab.left) (fun hb h => hb hab.right) (Eq.refl h)
-- The 'cases' became a recursor application. The script is not the term.
```

Search in the environment: exact?



```
example (a b : Nat) : a + b = b + a := by exact?      -- Try this: exact Nat.add_comm a b
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by exact?  -- Try this: exact ⟨hp, hq⟩
```

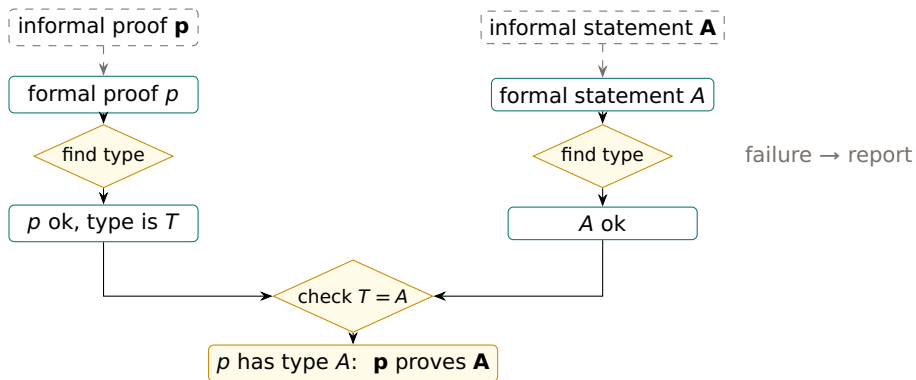
Why is a Lean proof trusted?



- 1 a term is a complete coding of its proof
- 2 Type Checking is decidable, from $\lambda \rightarrow$ up to λD
- 3 the **de Bruijn criterion**: any means may produce the term, as long as a small kernel checks it
- 4 the primitive constants a term rests on can be listed

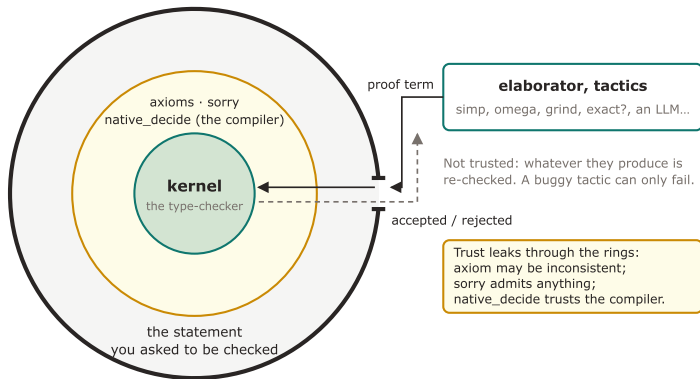
N&G Rem. 2.9.1
N&G Thm. 2.10.10, 6.3.15, 10.5.8
N&G §16.2
N&G Cor. 10.5.4

The de Bruijn criterion



Only the dashed arrows are not done by the computer; the final check is $\beta\delta$ -convertibility. **The de Bruijn criterion:** a proof assistant generates proof terms a small kernel can check, whatever produced them. N&G §16.2, Fig. 16.1

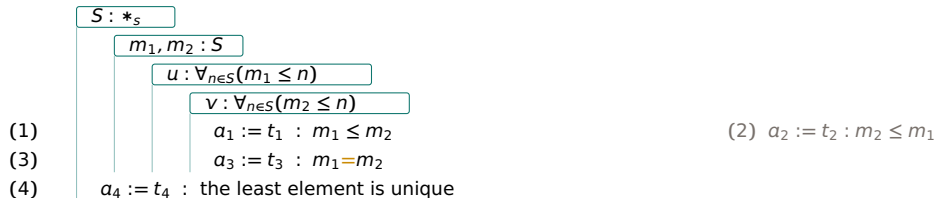
What you are trusting: the trusted computing base



The TCB is what must be right for a proof to mean anything — everything else may be wrong and is caught.

A first attempt at mathematics, and what it lacks

Lemma: a least element of a partially ordered set is unique. Its proof, as a flag derivation:



not yet in Δ

the symbol $=$ in line (3)
the term t_3
“ S has a least element”
“unique”
the term t_4

what must be defined

equality, as a notion in λD
how two $\leq$ give an $=$
sets and predicates, with $\exists$
 $\exists!$, from $=$ and $\exists$
a proof using all of the above

I The programme

Every gap is a definition still missing from Δ . Equality and sets come next lecture; numbers and induction after that.

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).

Bruijn, N. G. de (1970). “The mathematical language AUTOMATH, its usage, and some of its extensions”. In: *Symposium on Automatic Demonstration*. Vol. 125. Lecture Notes in Mathematics. Springer, pp. 29–61.

Diaconescu, R. (1975). “Axiom of choice and complementation”. In: *Proceedings of the American Mathematical Society* 51.1, pp. 176–178.

Nederpelt, R. and H. Geuvers (2014a). “Definitions”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 8, pp. 165–188.

— (2014b). “Extension of λC with definitions”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 9, pp. 189–210.

References II

Nederpelt, R. and H. Geuvers (2014c). “Flag-style natural deduction in λD ”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 11, pp. 225–256.

— (2014d). “Further perspectives”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 16, pp. 379–390.

— (2014e). “Rules and properties of λD ”. In: *Type Theory and Formal Proof: An Introduction*. Cambridge: Cambridge University Press. Chap. 10, pp. 211–224.

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

References III

Chapters 8 to 11 of the book throughout. Ch. 8: Def. 8.3.1, Rem. 8.3.2, §8.4 and Rem. 8.4.1, §8.5–8.8, Figs. 8.1–8.7, Not. 8.7.1. Ch. 9: Def. 9.1.2, 9.2.1, Rem. 9.2.2, 9.2.4, Def. 9.3.1, 9.4.7, 9.5.1, Rem. 9.5.3, Def. 9.5.5, 9.5.7, Fig. 9.1–9.2, Def. 9.7.2, 9.7.3, Fig. 9.3. Ch. 10: Def. 10.2.1, 10.4.2, Ex. 10.2.2, Fig. 10.1, Thm. 10.5.2, 10.5.3, Cor. 10.5.4, Thm. 10.5.5, Cor. 10.5.6, Thm. 10.5.8, §10.6. Ch. 11: §11.1–11.3 and Figs. 11.1–11.2, §11.4–11.5 and Figs. 11.5–11.12, §11.6–11.7 and Figs. 11.13–11.15, Not. 11.7.1, §11.8 and Figs. 11.16–11.17, Rem. 11.8.1, 11.9.3, §11.10 and Figs. 11.22–11.24, Rem. 11.10.2; Appendix A. §12.1 and Fig. 12.1 for the gap. §16.1–16.3 and Fig. 16.1 for trust; Rem. 14.1.3. The Lean slides follow *Theorem Proving in Lean 4*: §5.3–5.4 (structured scripts, obtain, exact?), ch. 6 (sections, namespaces, implicit arguments), §12.2–12.6 (the three primitive constants and what is derived from them).