“In mathematics, you don’t understand things. You just get used to them.”

— John von Neumann

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 2 of 10

Dependent Type Theory

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Google Developer Groups
On Campus • American University of Beirut

MATHEMATICS SOCIETY
AMERICAN UNIVERSITY OF BEIRUT

Section 1

Lambda Calculus: The Foundation

Why Lambda Calculus?

The Problem: Programming languages are complex with lots of syntax.

The Solution: Lambda calculus is a theory of functions with only:

- 3 pieces of syntax
- 1 rule of computation
- Ability to express *anything* computable

Lambda calculus gives us a mathematical foundation to reason about computation itself (and LARGELY predates modern programming languages).

Key point: Just like arithmetic has addition and multiplication as primitive operations, lambda calculus has *function application* as its only primitive operation.

Functions as Mappings

For our intents, a function is simply a mapping of inputs (domain) to outputs (codomain).

Example: $f(x) = x^2$ maps:

- $2 \mapsto 4$
- $3 \mapsto 9$
- $x \mapsto x^2$ (for arbitrary x)

Instead of writing $f(x) = x^2$ (like in math), in the Lambda Calculus, we would write: $\lambda x.\, x^2$

Read as: "lambda x maps to x^2"

Note: Nothing special about "λ" - could be "ballout", "jazar", or "douleb". The symbol is arbitrary!

Lambda Abstraction Syntax

Lambda abstraction: $\lambda x.\, M$

- λ indicates we're defining a function
- x is the input variable (parameter)
- . separates parameter from body
- M is the output expression (body)

Examples:

- $\lambda x.\, x + 1$ (successor function)
- $\lambda y.\, y \times y$ (square function)
- $\lambda n.\, \lambda m.\, n + m$ (addition, so two parameters!)
- $\lambda f.\, \lambda x.\, f(f(x))$ (apply function twice)

In Lean: `fun x => x + 1` or `λ x => x + 1`

Beta Reduction: Function Application

Beta reduction (β-reduction): Applying a function to an argument

Rule: $(\lambda x.\, M)\; N \to_\beta M[N/x]$
Meaning:

- $(\lambda x.\, M)$ is the function definition
- N is the argument being supplied
- $M[N/x]$ means: "in M, replace every x with N"

Simple example:

- $(\lambda y.\, y \times y)\; 5 \to_\beta 5 \times 5 = 25$

Beta Reduction: Complex Example

Let's evaluate: $(\lambda f.\, \lambda x.\, f(f(x)))\; (\lambda y.\, y + 1)\; 2$

Step-by-step reduction:

$$
\begin{aligned}
& (\lambda f.\, \lambda x.\, f(f(x)))\; (\lambda y.\, y + 1)\; 2 \\
&= (\lambda x.\, (\lambda y.\, y + 1)((\lambda y.\, y + 1)(x)))\; 2 && \text{(Apply first arg)} \\
&= (\lambda y.\, y + 1)((\lambda y.\, y + 1)(2)) && \text{(Apply second arg)} \\
&= (\lambda y.\, y + 1)(2 + 1) && \text{(Inner reduction)} \\
&= (\lambda y.\, y + 1)(3) && \text{(Arithmetic)} \\
&= 3 + 1 && \text{(Final reduction)} \\
&= 4 && \text{(Result)}
\end{aligned}
$$

Pro tip: Work from outside in ("leftmost-outermost" strategy)

More Beta Reduction Examples

Example 1: Identity function

$$(\lambda x.\, x)\; 5 \to_\beta 5$$

Example 2: Constant function

$$(\lambda x.\, \lambda y.\, x)\; 5\; 3 \to_\beta (\lambda y.\, 5)\; 3 \to_\beta 5$$

Example 3: Function composition

$$\begin{aligned} &(\lambda f.\, \lambda g.\, \lambda x.\, f(g(x)))\; (\lambda y.\, y \times 2)\; (\lambda z.\, z + 1)\; 3 \\ &= (\lambda y.\, y \times 2)((\lambda z.\, z + 1)(3)) \\ &= (\lambda y.\, y \times 2)(4) \\ &= 4 \times 2 = 8 \end{aligned}$$

Alpha Conversion: Renaming Variables

Alpha equivalence (α-equivalence): Functions are equivalent if they differ only in variable names

Examples (all equivalent):

- $\lambda x.\, x^2$
- $\lambda y.\, y^2$
- $\lambda z.\, z^2$
- $\lambda banana.\, banana^2$

The choice of variable name doesn't change what the function does!

Alpha Conversion: Why It Matters

Variable capture problem: Must rename to avoid conflicts

Bad substitution:

$$(\lambda x.\, \lambda y.\, x)\, y \to_\beta \lambda y.\, y \quad \text{(wrong)}$$

Correct substitution (with α-conversion):

$$\begin{aligned}(\lambda x.\, \lambda y.\, x)\, y &\to_\alpha (\lambda x.\, \lambda y'.\, x)\, y \\ &\to_\beta \lambda y'.\, y \quad \text{(correct)}\end{aligned}$$

Rule: Rename bound variables before substitution to avoid capture

Lambda Calculus in Lean

Lean uses lambda calculus as its foundation!

```
-- Lambda abstraction (two syntaxes)
#check fun x : Nat => x + 1
#check λ x : Nat => x * x

-- Function application
#eval (fun x => x + 1) 5  -- 6

-- Higher-order functions
def twice (f : Nat → Nat) (x : Nat) : Nat :=
  f (f x)

#eval twice (fun x => x + 1) 5  -- Result: 7
#eval twice (fun x => x * 2) 3  -- Result: 12
```

Currying in Lean

Currying: Multiple parameters via nested functions

```
-- Explicit currying
def add : Nat → Nat → Nat :=
  fun m ⇒ fun n ⇒ m + n

#eval add 3 4  -- Result: 7
#eval (add 3) 4  -- Result: 7 (same thing!)

-- Syntactic sugar (same as above)
def add' (m : Nat) (n : Nat) : Nat := m + n

-- Partial application
def add5 := add 5
#eval add5 10  -- Result: 15
```

Key benefit: Partial application lets us create specialized functions!

Higher-Order Functions

TL;DR: Functions can:

- Take functions as inputs
- Return functions as outputs

Example: $\lambda f.\, \lambda x.\, f(f(x))$

This expression takes a function f and returns a new function that applies f twice.

Practical examples:

- `map`: Apply function to every list element
- `filter`: Keep elements satisfying a predicate
- `compose`: Chain two functions together

Higher-Order Functions in Lean

Here are some ways to write higher-order functions in Lean for you to contemplate:

```
-- Function that applies f n times
def applyN (f : Nat → Nat) : Nat → Nat → Nat
  | 0, x => x
  | n+1, x => f (applyN f n x)

#eval applyN (· + 1) 5 0  -- Result: 5

-- Function composition
def compose (f : Nat → Nat) (g : Nat → Nat) : Nat → Nat :=
  fun x => f (g x)

def double := (· * 2)
def increment := (· + 1)
def doubleAndIncrement := compose increment double

#eval doubleAndIncrement 5  -- Result: 11 (5*2 + 1)
```

Currying: Multiple Arguments

Lambda abstractions only take *one* argument. How do we handle multiple inputs?

Currying: Return a function that takes the next argument

Example: Addition

$$add = \lambda x.\, \lambda y.\, x + y$$
$$add\, 1 \to_\beta \lambda y.\, 1 + y$$
$$(add\, 1)\, 2 \to_\beta 1 + 2 = 3$$

Named after logician **Haskell Curry**.

Key benefit: Partial application! `add 1` is a valid function.

Encoding Booleans

We can encode data types as functions! (Church encodings)

Church Booleans:

- $true = \lambda x.\, \lambda y.\, x$ (returns first argument)
- $false = \lambda x.\, \lambda y.\, y$ (returns second argument)

If-then-else:

$$if = \lambda b.\, \lambda x.\, \lambda y.\, b\, x\, y$$

If b is true, returns x; if false, returns y!

Church Booleans: How They Work

Recall:

- $true = \lambda x.\, \lambda y.\, x$
- $false = \lambda x.\, \lambda y.\, y$

Example evaluation:

$$\begin{aligned} true\; a\; b &= (\lambda x.\, \lambda y.\, x)\; a\; b \\ &\to_\beta (\lambda y.\, a)\; b \\ &\to_\beta a \end{aligned}$$

$$\begin{aligned} false\; a\; b &= (\lambda x.\, \lambda y.\, y)\; a\; b \\ &\to_\beta (\lambda y.\, y)\; b \\ &\to_\beta b \end{aligned}$$

Insight: Booleans are *choice functions*!

Boolean Operations

Boolean operations using Church encoding:

NOT:

$$not = \lambda p.\, p\, false\, true$$

AND:

$$and = \lambda p.\, \lambda q.\, p\, q\, p$$

If p is true, return q; if p is false, return p (false)

OR:

$$or = \lambda p.\, \lambda q.\, p\, p\, q$$

If p is true, return p (true); if p is false, return q

Boolean Operations: Example

Let's evaluate: $\mathit{and}\ \mathit{true}\ \mathit{false}$

$$
\begin{aligned}
&\mathit{and}\ \mathit{true}\ \mathit{false} \\
&= (\lambda p.\, \lambda q.\, p\, q\, p)\ \mathit{true}\ \mathit{false} \\
&= (\lambda q.\, \mathit{true}\ q\ \mathit{true})\ \mathit{false} \\
&= \mathit{true}\ \mathit{false}\ \mathit{true} \\
&= (\lambda x.\, \lambda y.\, x)\ \mathit{false}\ \mathit{true} \\
&= (\lambda y.\, \mathit{false})\ \mathit{true} \\
&= \mathit{false}
\end{aligned}
$$

Result: false (as expected!)

Encoding Numbers (Church Numerals)

We can even encode natural numbers as functions!

Church Numerals: A number n is a function that applies f exactly n times

- $0 = \lambda f.\, \lambda x.\, x$ (apply f zero times)
- $1 = \lambda f.\, \lambda x.\, f\, x$ (apply f once)
- $2 = \lambda f.\, \lambda x.\, f\,(f\, x)$ (apply f twice)
- $3 = \lambda f.\, \lambda x.\, f\,(f\,(f\, x))$ (apply f three times)

Insight: A number is an *iterator*!

Church Numerals: Successor

Successor function: Add one to a Church numeral

$$succ = \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\; f\; x)$$

How it works:

- Take a number n (which applies f n times)
- Apply f to the result of $n\; f\; x$
- This gives us $n + 1$ applications of f

Example: $succ\, 2$

$$\begin{aligned} succ\, 2 &= (\lambda n.\, \lambda f.\, \lambda x.\, f\,(n\; f\; x))\,(\lambda f.\, \lambda x.\, f\,(f\, x)) \\ &\to_\beta \lambda f.\, \lambda x.\, f\,((\lambda f'.\, \lambda x'.\, f'\;(f'\; x'))\; f\, x) \\ &\to_\beta \lambda f.\, \lambda x.\, f\,(f\,(f\, x)) = 3 \end{aligned}$$

Church Numerals: Addition

Addition: Add two Church numerals

$$add = \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\, f\, (n\, f\, x)$$

How it works:

- Apply f n times to x (giving us n)
- Then apply f m more times
- Total: $m + n$ applications of f

Example: $add\, 2\, 3 = 5$

- $2\, f\, (3\, f\, x) = f\, (f\, (f\, (f\, (f\, x)))) = 5$

Church Numerals: Multiplication

Multiplication: Multiply two Church numerals

$$mult = \lambda m.\, \lambda n.\, \lambda f.\, m\,(n\; f)$$

How it works:

- $n\; f$ creates a function that applies f n times
- $m\,(n\; f)$ applies this function m times
- Total: $m \times n$ applications of f

Example: $mult\; 2\; 3$

- Apply "f three times" twice
- $= f\,(f\,(f\,(f\,(f\,(f\,x))))) = 6$

Simply Typed Lambda Calculus

Problem: Nothing stops nonsense like:

- Applying NOT to a number
- Adding a boolean to a string
- Using 42 as a function

Solution: Add a *type system*

Assign types to terms:

- $true : \text{Bool}$
- $3 : \text{Nat}$
- $\lambda x : \text{Bool}.\ not\ x : \text{Bool} \to \text{Bool}$
- $\lambda x : \text{Nat}.\ \lambda y : \text{Nat}.\ x + y : \text{Nat} \to \text{Nat} \to \text{Nat}$

Type System Rules

Type checking rule: Can only apply $f : A \to B$ to arguments of type A

Valid applications:

- $(\lambda x : \text{Nat}.\ x + 1) : \text{Nat} \to \text{Nat}$
- Apply to $5 : \text{Nat}$ (correct)
- Result: $6 : \text{Nat}$

Invalid applications:

- Apply to $true : \text{Bool}$ (wrong)
- Type error: expected Nat, got Bool

Why Do We Care About Types?

Types catch errors at compile time:

- $(\lambda x : \text{Nat}.\ x + 1)\ 5$ (correct) (type checks)
- $(\lambda x : \text{Nat}.\ x + 1)\ \mathit{true}$ (wrong) (type error!)

Types are documentation:

- map : $(A \to B) \to \text{List}\ A \to \text{List}\ B$
- Type signature tells us what the function does!

Types enable optimization:

- Compiler knows exact memory layout
- Can inline functions safely
- Enables aggressive optimizations

Types as Specifications

In dependent type theory: Types can express correctness properties

Examples:

- `Vector Nat 5`: a list of exactly 5 natural numbers
- `sort : List Nat → {xs : List Nat // xs.Sorted}`: returns a sorted list
- `safeDiv : (n : Nat) → (d : Nat) → (d ≠ 0) → Nat`: division requires proof denominator is non-zero

Motto: "If it compiles, it's probably correct"

Curry-Howard Correspondence (1)

As it turns out, there's a deep connection between Type Theory and Logic:

$$
\begin{array}{rcl}
\textbf{Programs} & \leftrightarrow & \textbf{Proofs} \\
\textbf{Types} & \leftrightarrow & \textbf{Propositions} \\
\textbf{Terms} & \leftrightarrow & \textbf{Proofs} \\
\rightarrow & \leftrightarrow & \implies
\end{array}
$$

Examples:

- Type $A \rightarrow B \cong$ Proposition "A implies B"
- Term of type $A \rightarrow B \cong$ *Proof* of "$A \implies B$"
- Type checking $\cong$ Proof checking!

Curry-Howard: More Correspondence Examples

Extended correspondences:

Product type $A \times B$	$\leftrightarrow$	**Conjunction** $A \wedge B$
Sum type $A + B$	$\leftrightarrow$	**Disjunction** $A \vee B$
Empty type	$\leftrightarrow$	**False**
Unit type	$\leftrightarrow$	**True**
Type inhabitation	$\leftrightarrow$	**Provability**

Key point: A proof is a program, and vice versa!

Curry-Howard: Modus Ponens Example

Logic: If we have $A \implies B$ and A, we can derive B

As a function:

```
-- The type is the proposition
def modus_ponens {A B : Prop} (h1 : A → B) (h2 : A) : B :=
  h1 h2  -- Apply the implication to the hypothesis
```

Observations:

- Type signature = Logical statement
- Function body = Proof
- Type checking = Proof verification

Curry-Howard: Hypothetical Syllogism Example

Logic: $(A \implies B) \implies (B \implies C) \implies (A \implies C)$

As a function:

```
def chain {A B C : Prop} (f : A → B) (g : B → C) : A → C :=
  fun h : A ⇒ g (f h)
```

This is just function composition!

- Logical proof = Function composition
- Proving theorems = Writing programs

Key point: Writing programs = Constructing proofs!

Curry-Howard: Digression on Uninhabited Types

Empty type: A type with no values

In logic: Corresponds to False

Key property: From False, anything follows (ex falso quodlibet)

If we have a term of type `Empty`, we can construct a term of *any* type:

$$\text{absurd} : \text{Empty} \to A$$

Why? Because we can never actually call this function (no terms of type `Empty` exist)!

From the Lambda Calculus to Dependent Types

The Lambda Calculus gave us:

- Functions (λ-abstractions)
- Function application (β-reduction)
- Higher-order functions

Simply Typed Lambda Calculus added:

- Type system for safety
- Type checking
- Curry-Howard correspondence

But we can go further...

The Limitation of Simple Types

In simple types, we have things like:

- `List Nat`: list of natural numbers
- `List String`: list of strings
- `List Bool`: list of booleans

But these types don't tell us:

- How many elements in the list?
- Is the list sorted?
- Are all elements positive?

(Non-trivial) Solution: Let types depend on values!

What If Types Could Depend on Values?

Dependent types: Types that depend on values (as the name implies)

Examples:

- `Vector Nat 3`: a list of exactly 3 natural numbers
- `Vector Nat n`: a list of exactly n natural numbers
- `Matrix m n`: an $m \times n$ matrix
- `Fin n`: natural numbers less than n

This is **Dependent Type Theory**!

Section 2

Types

Types in Lean

Types σ, τ, υ:

- Type variables: α, β, γ
- Basic types: `Nat`, `Int`, `Bool`, `String`
- Complex types: $T\,\sigma_1\ \ldots\ \sigma_N$ (e.g. List (Option Nat), but don't worry about it for now)

Some type constructors are written infix: $\rightarrow$ (function type)

Function arrow is right-associative:

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \tau = \sigma_1 \rightarrow (\sigma_2 \rightarrow (\sigma_3 \rightarrow \tau))$$

Polymorphic types use type variables:

```
#check fun {α : Type} (x : α) ⇒ x  -- id : α → α (type)
#check List  -- Type → Type (type constructor)
```

Type Examples (in Lean)

Types indicate which values an expression may evaluate to.

```
#check ℕ          -- Type (natural numbers)
#check ℤ          -- Type (integers)
#check Empty      -- Type (no values, False)
#check Unit       -- Type (one value, trivial type)
#check Bool       -- Type (true and false)

-- Function types
#check ℕ → ℤ                  -- Nat to Int
#check ℤ → ℕ                  -- Int to Nat (partial!)
#check Bool → ℕ → ℤ           -- Bool → (ℕ → ℤ)
#check (Bool → ℕ) → ℤ         -- Different since it takes a function
#check ℕ → (Bool → ℕ) → ℤ     -- Explicit parentheses
```

More Type Examples (in Lean)

```
-- Polymorphic types
#check List ℕ                -- List of natural numbers
#check List (List String)    -- List of lists of strings
#check α → α                 -- Generic identity function type

-- Function types with multiple arrows
#check Nat → Nat → Nat       -- Two arguments, one result
#check (Nat → Nat) → Nat     -- Takes function as argument
#check Nat → (Nat → Nat)     -- Returns a function
```

Key point: Parentheses matter!

- `Nat → Nat → Nat` = `Nat → (Nat → Nat)`
- `(Nat → Nat) → Nat` is different (higher-order)

Type Constructors

Type constructors build new types from existing ones

```
-- List is a type constructor: Type → Type
#check List      -- Type → Type
#check List Nat  -- Type

-- Product types (tuples)
#check Nat × String           -- Type
#check (3, "hello")           -- Nat × String

-- Sum types (disjoint union)
#check Nat ⍰ String           -- Type
#check Sum.inl 42             -- Nat ⍰ String
#check Sum.inr "hi"           -- Nat ⍰ String
```

Option Type

Option: Represents a value that may or may not exist

```
-- Option type (maybe)
#check Option Nat              -- Type
#check some 5                  -- Option Nat (has a value)
#check none                    -- Option Nat (no value)

-- Useful for partial functions
def safeHead (xs : List Nat) : Option Nat :=
  match xs with
  | [] ⇒ none
  | x :: _ ⇒ some x

#eval safeHead [1, 2, 3]  -- some 1
#eval safeHead []         -- none
```

Key benefit: No null pointer exceptions!

Product Types (Pairs)

Product type $A \times B$: Contains a value of type A AND a value of type B

Examples:

- `Nat × String`: a number and a string
- `(3, "hello") : Nat × String`
- `Bool × Bool × Bool`: three booleans

Access components:

- `p.1`: the first component
- `p.2`: the second component

In logic: Corresponds to conjunction ($A \wedge B$)

Sum Types (Disjoint Union)

Sum type $A + B$: Contains a value of type A OR a value of type B

Constructors:

- `Sum.inl : A → A + B` (left injection)
- `Sum.inr : B → A + B` (right injection)

Examples:

- `Sum.inl 5 : Nat + String`: a number
- `Sum.inr ”hello” : Nat + String`: a string

In logic: Corresponds to disjunction ($A \vee B$)

Section 3

Terms

Terms

The terms (expressions) of type theory:

- **Constants**: c (built-in or defined values)
- **Variables**: x (parameters or bound variables)
- **Applications**: $t\ u$ (function applied to argument)
- **Lambda abstractions**: `fun x ↦ t` (anonymous functions)

Application is left-associative:

$$\texttt{f x y z} = \texttt{((f x) y) z}$$

Use `#check` to see the type of any term!

Term Examples

Consider how Lean allows us to construct simple lambda abstractions, abstractions with multiple (curried) parameters, and higher-order logic.

```
-- Simple lambda abstractions
#check fun x : ℕ ⇒ x                    -- ℕ → ℕ
#check fun (x : ℕ) ⇒ x + 1              -- ℕ → ℕ

-- Multiple parameters (curried)
#check fun (x y : ℕ) ⇒ x + y            -- ℕ → ℕ → ℕ

-- Higher-order functions
#check fun f : ℕ → ℕ ⇒ fun g : ℕ → ℕ ⇒
  fun h : ℕ → ℕ ⇒ fun x : ℕ ⇒ h (g (f x))

-- More concise (type inference!)
#check fun (f g h : ℕ → ℕ) (x : ℕ) ⇒ h (g (f x))
```

Note: functions are treated as "first-class values"; we don't need to annotate every intermediate type!

Type Inference

Type inference: Lean can often figure out types automatically!
We will soon cover in detail exactly how Lean does that.

```
-- Explicit types
#check fun (x : Nat) ⇒ x + 1  -- Nat → Nat

-- Inferred types
#check fun x ⇒ x + 1          -- Nat → Nat (inferred!)

-- Fully polymorphic
#check fun x ⇒ x              -- α → α (works for any type!)

-- Context helps inference
def double (n : Nat) := n * 2
#check fun x ⇒ double x       -- Nat → Nat (from double's type)
```

Best practice: Omit types when clear, add them for clarity

Opaque Constants and Axioms

After the `opaque` commands, we have no information about a and b beyond their type.

```
-- Opaque constants (axioms without proofs)
opaque a : ℤ
opaque b : ℤ
opaque f : ℤ → ℤ
opaque g : ℤ → ℤ → ℤ

#check fun x : ℤ ⇒ g (f (g a x)) (g x b)
#check fun x ⇒ g (f (g a x)) (g x b)  -- Type inferred
#check fun x ⇒ x                      -- Fully polymorphic
```

Opaque: Declared but not defined (typically used in conjuction with axioms)

For example:

```
opaque a : ℤ
opaque b : ℤ
axiom a_less_b :
    a < b
```

Section 4

Type Checking and Type Inference

Type Checking and Type Inference

Two key problems in type theory:

Type Checking: Given term t and type σ, does $t : \sigma$?

- Decidable for simple type theory
- Lean's kernel checks all proofs
- Ensures logical soundness

Type Inference: Given term t, find its type σ

- Also decidable for simple types
- Lets us omit type annotations
- Makes code more readable

Type Judgments

Type judgment: $C \vdash t : \sigma$

Read: "In context C, term t has type σ"

Context C: Tracks local variable types

- $C = \{x_1 : \sigma_1, x_2 : \sigma_2, ...\}$
- Variables and their types
- Built up during type checking

Example:

$$\{x : \text{Nat}, y : \text{Bool}\} \vdash x + 1 : \text{Nat}$$

Typing Rules

Rules are written as a fraction. In formal logic, these are **inference** rules: if the stuff on top (the premise) is true, then the stuff on the bottom (the conclusion) is also true.

Constant rule:

$$\frac{}{C \vdash c : \sigma}\text{Cst} \quad \text{if } c \text{ is declared with type } \sigma$$

Variable rule:

$$\frac{}{C \vdash x : \sigma}\text{Var} \quad \text{if } x : \sigma \in C$$

Application rule:

$$\frac{C \vdash t : \sigma \to \tau \quad C \vdash u : \sigma}{C \vdash t\, u : \tau}\text{App}$$

Abstraction rule:

$$\frac{C, x : \sigma \vdash t : \tau}{C \vdash (\text{fun } x : \sigma \mapsto t) : \sigma \to \tau}\text{Fun}$$

Type Checking Examples (1)

Check: $\vdash (\lambda x : \text{Nat}.\, x + 1)\, 5 : \text{Nat}$

Step 1: Check the function

- In context $C = \{x : \text{Nat}\}$
- Body $x + 1$ has type Nat
- By Abstraction rule: $\vdash \lambda x : \text{Nat}.\, x + 1 : \text{Nat} \to \text{Nat}$

Step 2: Check the application

- Function: $\text{Nat} \to \text{Nat}$
- Argument: $5 : \text{Nat}$
- By Application rule: Result type is Nat (correct)

Type Checking Examples (2)

Check: $\{f : \text{Nat} \to \text{Bool}\} \vdash f\,5 : \text{Bool}$

Given:

- Context: $C = \{f : \text{Nat} \to \text{Bool}\}$
- Term: $f\,5$

Type checking:

- $C \vdash f : \text{Nat} \to \text{Bool}$ (by Variable rule)
- $C \vdash 5 : \text{Nat}$ (by Constant rule)
- $C \vdash f\,5 : \text{Bool}$ (by Application rule) (correct)

Type Checking Examples (3)

Check: $\vdash (\lambda f : \text{Nat} \to \text{Nat}.\ \lambda x : \text{Nat}.\ f\,(f\,x)) : ?$

Step 1: Check inner lambda

- Context: $C = \{f : \text{Nat} \to \text{Nat}, x : \text{Nat}\}$
- Body: $f\,(f\,x)$ has type Nat
- Inner lambda: Nat $\to$ Nat

Step 2: Check outer lambda

- Context: $C = \{f : \text{Nat} \to \text{Nat}\}$
- Body: $\lambda x : \text{Nat}.\ f\,(f\,x)$ has type Nat $\to$ Nat
- Outer lambda: $(\text{Nat} \to \text{Nat}) \to (\text{Nat} \to \text{Nat})$ (correct)

Sidenote: Variable Shadowing

Shadowing: Inner variables hide outer ones with the same name

Example:

$$\lambda x : \mathsf{Nat}.\ \lambda x : \mathsf{Bool}.\ x$$

Which x? The inner one (Bool)!

Context tracking:

- Outer context: $\{x : \mathsf{Nat}\}$
- Inner context: $\{x : \mathsf{Nat}, x : \mathsf{Bool}\}$
- Rightmost occurrence shadows: $x : \mathsf{Bool}$

Best practice: Avoid shadowing (confusing!)

Section 5

Type Inhabitation

Type Inhabitation

Type Inhabitation Problem: Given a type σ, find a term of that type

Key fact: This problem is *undecidable* in general!

- Some types have no inhabitants (like `Empty`)
- Finding inhabitants = constructing proofs
- By Curry-Howard: Finding proof = Solving halting problem

Why do we care? Lean's `exact?` tactic tries this! (don't worry about this too much, we will get more into tactics later)

Type Inhabitation: Strategy

Recursive procedure (doesn't always terminate):

1. If $\sigma = \tau \to \upsilon$, try `fun x ⇒ _`
2. Look for constants/variables $c : \tau_1 \to \cdots \to \tau_N \to \sigma$
3. Build term c _ … _ and recursively fill holes

Example: Inhabit Nat $\to$ Nat

- Try: `fun x ⇒ _`
- Look for: variable x : Nat
- Solution: `fun x ⇒ x` (identity function)

Type Inhabitation Example

Problem: Inhabit $(\alpha \to \beta \to \gamma) \to ((\beta \to \alpha) \to \beta) \to \alpha \to \gamma$

```
opaque α : Type
opaque β : Type
opaque γ : Type

def someFunOfType : (α → β → γ) → ((β → α) → β) → α → γ :=
  fun f g a ⇒ f a (g (fun b ⇒ a))
  -- f : α → β → γ
  -- g : (β → α) → β
  -- a : α
  -- Need to produce: γ
  --
  -- f needs: α and β
  -- We have: a : α
  -- For β: use g applied to (fun b ⇒ a)
  -- Result: f a (g (fun b ⇒ a)) : γ (correct)
```

Type Inhabitation: More Examples (1)

Example 1: $A \to A$

- Solution: ?

Example 2: $A \to B \to A$

- Solution: ? (hint: constant)

Example 3: $(A \to B) \to (B \to C) \to (A \to C)$

- Solution: ? (hint: composition)

Example 4: Empty $\to A$

- Solution: ? (hint: absurd)

Type Inhabitation: More Examples (2)

Example 1: $A \to A$

- Solution: `fun x ⇒ x` (identity)

Example 2: $A \to B \to A$

- Solution: `fun x y ⇒ x` (const)

Example 3: $(A \to B) \to (B \to C) \to (A \to C)$

- Solution: `fun f g x ⇒ g (f x)` (composition)

Example 4: Empty $\to A$

- Solution: `fun x ⇒ match x with .` (absurd)

For Culture: Uninhabited Types

Some types have no inhabitants!

Examples:

- `Empty`: no constructors (since nothing can construct it)
- $A \to$ `Empty` where A is inhabited: there is no way to produce `Empty`
- $(A \to B) \to A$ in intuitionistic logic (Peirce's law)

In logic: Corresponds to unprovable propositions

- `Empty` = False statement or logical contradiction
- `Empty` $\to A$: "I can produce a value of any type A as long as you give me a value of Empty" (impossible!)
- Uninhabited type = Unprovable proposition

Section 6

Type Definitions

Inductive Types: The Foundation

Inductive type: A type consisting of all values built using its **constructors**, and *nothing else*

General format:

```
inductive TypeName (params : Types) : Type where
  | constructor1 : constructor1_type
  | constructor2 : constructor2_type
  | ...
  | constructorN : constructorN_type
```

Key properties:

- **No junk**: Only values from constructors exist
- **No confusion**: Different constructors $\neq$ different values
- **Finite**: No infinite chains of constructors

Natural Numbers: Peano Arithmetic (in Lean)

Lean's most fundamental inductive type:

```
namespace MyNat

inductive Nat : Type where
  | zero : Nat           -- Base case: 0
  | succ : Nat → Nat     -- Recursive: successor

-- How numbers are represented:
-- 0 = Nat.zero
-- 1 = Nat.succ Nat.zero
-- 2 = Nat.succ (Nat.succ Nat.zero)
-- 3 = Nat.succ (Nat.succ (Nat.succ Nat.zero))

#check Nat
#check Nat.zero
#check Nat.succ
#check Nat.succ (Nat.succ Nat.zero)  -- This is 2!

end MyNat
```

Why unary? Makes induction and recursion natural!

Peano Axioms

For Culture: Peano's axioms for natural numbers:

1. 0 is a natural number
2. Every natural number has a successor
3. 0 is not the successor of any number
4. Different numbers have different successors (injective)
5. Induction principle holds

How do we do this in Lean?

- `Nat.zero` gives us axiom (1)
- `Nat.succ` gives us axiom (2)
- "No confusion" gives us axioms (3) and (4)
- Pattern matching gives us axiom (5)

Arithmetic Expressions: Abstract Syntax Trees (in Lean)

Inductive types naturally represent syntax trees!

This is extremely useful for compiler/interpreter design

```
inductive AExp : Type where
  | num : ℤ → AExp                      -- Literal number
  | var : String → AExp                 -- Variable name
  | add : AExp → AExp → AExp            -- e1 + e2
  | sub : AExp → AExp → AExp            -- e1 - e2
  | mul : AExp → AExp → AExp            -- e1 * e2
  | div : AExp → AExp → AExp            -- e1 / e2

-- Example: (x + 3) * (y - 2)
def example_expr : AExp :=
  AExp.mul
    (AExp.add (AExp.var "x") (AExp.num 3))
    (AExp.sub (AExp.var "y") (AExp.num 2))

#check example_expr  -- AExp
```

Lists: Polymorphic Recursive Types

Definition: A "polymorphic recursive type" is a type that **can take other types as parameters** (like generics), allowing functions or data structures to operate on different types in a flexible manner. It is the workhorse of functional programming:

```
-- Conceptual definition (List is built-in)
inductive MyList (α : Type) where
  | nil  : MyList α                    -- Empty list
  | cons : α → MyList α → MyList α     -- Head :: Tail

-- Notation: [1, 2, 3] desugars to:
-- List.cons 1 (List.cons 2 (List.cons 3 List.nil))

#check List ℕ                -- Type
#check List.nil              -- List α
#check List.cons 1 List.nil  -- List ℕ  (the list [1])

-- Polymorphism: Works for ANY type α!
#check ([1, 2, 3] : List Nat)
#check (["a", "b"] : List String)
#check ([[1], [2, 3]] : List (List Nat))
```

Binary Trees

Recall: A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the “left child” and the “right child”.

```
inductive BTree (α : Type) : Type where
  | empty : BTree α
  | node : BTree α → α → BTree α → BTree α

-- Example tree:
--        5
--       / \
--      3   7
--     /
--    1
def exampleTree : BTree Nat := -- Explicitly write out the binary tree using its constructors
  BTree.node
    (BTree.node (BTree.node BTree.empty 1 BTree.empty) 3 BTree.empty)
    5
    (BTree.node BTree.empty 7 BTree.empty)
```

Uses: Search trees, sorting, expression trees, game trees, ...

Result Type (Either)

Result: Represents success or failure

Cool consequence: we can encode program state **using the type system itself**!

```
inductive Result (α : Type) : Type where
  | success : α → Result α
  | failure : String → Result α

-- Example: Safe division
def safeDiv (n : Nat) (d : Nat) : Result Nat :=
  if d = 0 then
    Result.failure "Division by zero"
  else
    Result.success (n / d)

#eval safeDiv 10 2  -- success 5
#eval safeDiv 10 0  -- failure "Division by zero"
```

Benefit: Explicit error handling in types!

Section 7

Function Definitions

Pattern Matching: Defining Functions

Fact: we can define functions by **pattern matching** on constructors:

```
-- Fibonacci numbers
def fib : ℕ → ℕ
  | 0     ⇒ 0
  | 1     ⇒ 1
  | n + 2 ⇒ fib (n + 1) + fib n

#eval fib 10  -- Result: 55

-- Addition (recursive on second argument)
def add : ℕ → ℕ → ℕ
  | m, Nat.zero   ⇒ m
  | m, Nat.succ n ⇒ Nat.succ (add m n)

#eval add 3 4  -- Result: 7
```

Lean verifies: Patterns are exhaustive and terminating!

Pattern Matching: The n+k Pattern

Special pattern: `n + k` matches numbers $\geq k$

Example: `n + 2`

- Matches: $2, 3, 4, 5, \ldots$
- Binds: $n = 0, 1, 2, 3, \ldots$
- Does not match: $0, 1$

Usage in Fibonacci:

- `fib 0 ⇒ 0`
- `fib 1 ⇒ 1`
- `fib (n + 2) ⇒ fib (n + 1) + fib n`

This is more elegant than explicit `succ` patterns!

Pattern Matching: Multiple Arguments

You can also very easily perform pattern matching on multiple arguments!

```
-- Pattern match multiple arguments
def min : ℕ → ℕ → ℕ
  | 0, _          ⇒ 0
  | _, 0          ⇒ 0
  | n+1, m+1      ⇒ (min n m) + 1

#eval min 3 5  -- Result: 3
#eval min 5 3  -- Result: 3

-- isEven using n+2 pattern
def isEven : ℕ → Bool
  | 0     ⇒ true
  | 1     ⇒ false
  | n + 2 ⇒ isEven n

#eval isEven 0   -- Result: true
#eval isEven 1   -- Result: false
#eval isEven 10  -- Result: true
#eval isEven 15  -- Result: false
```

Structural Recursion

Structural recursion: Recursion that "peels off" constructors

```
-- Length of a list (structurally recursive)
def length {α : Type} : List α → Nat
  | []      ⇒ 0
  | x :: xs ⇒ 1 + length xs  -- Perform recursion on xs (smaller!)

#eval length [1, 2, 3, 4]  -- Result: 4

-- Append two lists
def append {α : Type} : List α → List α → List α
  | [],      ys ⇒ ys
  | x :: xs, ys ⇒ x :: append xs ys  -- Perform recursion on xs

#eval append [1, 2] [3, 4, 5]  -- Result: [1, 2, 3, 4, 5]
```

Pedantic note: It is a generalization of mathematical induction to arbitrary inductive types. To prove a goal $n : \mathbb{N} \vdash P[n]$ by structural induction on n, it suffices to show two subgoals:

$$\vdash P[0] \;\; \text{(base case)}$$

$$k : \mathbb{N}, ih : P[k] \vdash P[k+1] \;\; \text{(induction step)}$$

More List Functions

Consider some very important list functions: **reverse**, **map** (apply function to each element), and **filter** (keep elements satisfying predicate).

```
-- Reverse a list
def reverse {α : Type} : List α → List α
  | []      ⇒ []
  | x :: xs ⇒ reverse xs ++ [x]

#eval reverse [1, 2, 3]  -- Result: [3, 2, 1]

-- Map: Apply function to each element
def listMap {α β : Type} (f : α → β) : List α → List β
  | []      ⇒ []
  | x :: xs ⇒ f x :: listMap f xs

#eval listMap (· * 2) [1, 2, 3]  -- Result: [2, 4, 6]

-- Filter: Keep elements satisfying predicate
def listFilter {α : Type} (p : α → Bool) : List α → List α
  | []      ⇒ []
  | x :: xs ⇒ if p x then x :: listFilter p xs else listFilter p xs
```

Note: These functions are vital in functional programming and you will use them all the time!

Fold: The Universal List Function

Fold: Process a list to produce a single value

Intuition: It literally "folds" the list into a value (right to left or left to right)

```
-- Left fold
def foldl {α β : Type} (f : β → α → β) (init : β) : List α → β
  | []      ⇒ init
  | x :: xs ⇒ foldl f (f init x) xs

-- Right fold
def foldr {α β : Type} (f : α → β → β) (init : β) : List α → β
  | []      ⇒ init
  | x :: xs ⇒ f x (foldr f init xs)

-- Examples
#eval foldl (· + ·) 0 [1, 2, 3, 4]  -- Result: 10 (sum)
#eval foldr (· :: ·) [] [1, 2, 3]   -- Result: [1, 2, 3] (identity)
#eval foldl (fun acc x ⇒ x :: acc) [] [1, 2, 3]  -- Result: [3, 2, 1] (reverse)
```

Fold Left vs Fold Right

Key difference: Order of operations

foldl (left fold): $(((z \circ x_1) \circ x_2) \circ x_3)$

- Tail recursive (efficient)
- Associates to the left
- Example: `foldl (-) 10 [1,2,3]` $= ((10 - 1) - 2) - 3 = 4$

foldr (right fold): $(x_1 \circ (x_2 \circ (x_3 \circ z)))$

- Not tail recursive
- Associates to the right
- Example: `foldr (-) 10 [1,2,3]` $= 1 - (2 - (3 - 10)) = -8$

Named Arguments

The colon (:) acts as a boundary (def mul … $: \mathbb{N} \to \mathbb{N}$): arguments placed to its left are fixed parameters available throughout the function, while those to its right are the inputs being analyzed via pattern matching.

```
-- Parameter m doesn't need pattern matching
def mul (m : ℕ) : ℕ → ℕ
  | Nat.zero   ⇒ Nat.zero
  | Nat.succ n ⇒ add m (mul m n)  -- m is in scope!

#eval mul 3 4  -- Result: 12

-- Generic iterator (higher-order function)
def iter (α : Type) (z : α) (f : α → α) : ℕ → α
  | Nat.zero   ⇒ z
  | Nat.succ n ⇒ f (iter α z f n)

-- Exponentiation using iter
def power (m n : ℕ) : ℕ :=
  iter ℕ 1 (mul m) n

#eval power 2 10  -- Result: 1024
```

Implicit Type Parameters

```
-- Type parameters can be implicit!
def reverse {α : Type} : List α → List α
  | []      ⇒ []
  | x :: xs ⇒ reverse xs ++ [x]

-- Lean infers α automatically
#eval reverse [1, 2, 3]        -- α = Nat (inferred!)
#eval reverse ["a", "b", "c"]  -- α = String (inferred!)

-- Can make explicit with @
#eval @reverse Nat [1, 2, 3]
```

Syntax:

- `{α : Type}` – Implicit (inferred)
- `(α : Type)` – Explicit (must provide)

Evaluating Expressions

First, define a way to look up the values of strings. In Lean, a common way to represent this is as a function String $\to \mathbb{Z}$.

```
def Env := String → ℤ -- An environment maps variable names to their integer values
def my_env : Env := fun name ⇒ -- Example environment where x = 10 and y = 5
    if name = "x" then 10
    else if name = "y" then 5
    else 0 -- Default value for unknown variables
```

You can use **pattern matching** on the inductive type. This is the functional way to "extract" the data from the constructors!

```
def eval (env : Env) : AExp → ℤ
 | AExp.num i     ⇒ i                    -- Literal
 | AExp.var x     ⇒ env x                -- Lookup variable
 | AExp.add e₁ e₂ ⇒ eval env e₁ + eval env e₂
 | AExp.sub e₁ e₂ ⇒ eval env e₁ - eval env e₂
 | AExp.mul e₁ e₂ ⇒ eval env e₁ * eval env e₂
 | AExp.div e₁ e₂ ⇒ eval env e₁ / eval env e₂
```

Evaluator Example

```
-- Example environment
def myEnv : String → ℤ
  | "x" ⇒ 5
  | "y" ⇒ 3
  | _   ⇒ 0

-- (x + 3) * (y - 2)  where x=5, y=3
def myExpr : AExp :=
  AExp.mul
    (AExp.add (AExp.var "x") (AExp.num 3))
    (AExp.sub (AExp.var "y") (AExp.num 2))

#eval eval myEnv myExpr  -- Result: (5+3)*(3-2) = 8
```

Key point: Pattern matching + recursion = **an actual (simple) interpreter**!

Termination Checking

Lean only accepts functions proven to terminate!

Why? Non-terminating functions can prove False:

- If `loop : ℕ → ℕ` where `loop n = loop n + 1`
- Then `loop 0 = loop 0 + 1`
- Subtract both sides: `0 = 1`
- From contradiction, anything follows! (Logical inconsistency)

What Lean accepts:

- **Structural recursion**: Recursive calls on structurally smaller arguments
- **Well-founded recursion**: Recursive calls on ”smaller” arguments (custom ordering)
- Mutual recursion (multiple functions calling each other)

Why Do We Care About Termination?

Example of inconsistency:

Suppose we allowed:

- `def loop (n : Nat) : Nat := loop n + 1`

Then:

$$
\begin{aligned}
\text{loop}\,0 &= \text{loop}\,0 + 1 \\
\text{loop}\,0 &= (\text{loop}\,0 + 1) + 1 \\
\text{loop}\,0 &= \text{loop}\,0 + 2 \\
&\vdots \\
0 &= n \text{ for any } n
\end{aligned}
$$

Disaster: We can prove anything! The logic is broken.

What Lean Rejects

Consider a couple of examples of functions Lean will reject.

```
-- (wrong) Not structurally smaller
def bad (n : ℕ) : ℕ := bad n + 1

-- (wrong) Growing, not shrinking
def worse (n : ℕ) : ℕ := worse (n + 1)

-- (wrong) Not obviously decreasing
def tricky (n : ℕ) : ℕ :=
  if n = 0 then 0 else tricky (n - 1 + 1)

-- (correct) Structurally decreasing
def good (n : ℕ) : ℕ :=
  if n = 0 then 0 else good (n - 1)
```

Lean's termination checker: Very smart, but not perfect!

Section 8

Theorem Statements

Theorems: Propositions as Types

Theorem: Like `def`, but result is a *proposition*

```
-- Commutativity of addition
theorem add_comm (m n : ℕ) :
  add m n = add n m :=
  sorry  -- Proof to be filled

-- Associativity of addition
theorem add_assoc (l m n : ℕ) :
  add (add l m) n = add l (add m n) :=
  sorry

-- Multiplication commutes
theorem mul_comm (m n : ℕ) :
  mul m n = mul n m :=
  sorry
```

`sorry` is a placeholder that assumes the proposition (unsafe!).

More Theorems

```
-- Reverse is involutive
theorem reverse_reverse {α : Type} (xs : List α) :
  reverse (reverse xs) = xs :=
  sorry

-- Append is associative
theorem append_assoc {α : Type} (xs ys zs : List α) :
  append (append xs ys) zs = append xs (append ys zs) :=
  sorry

-- Length of append
theorem length_append {α : Type} (xs ys : List α) :
  length (append xs ys) = add (length xs) (length ys) :=
  sorry
```

Later: We'll learn to write actual proofs!

Theorems vs Definitions

What's the difference?

Definition (`def`):

- Defines a computational function
- Can be evaluated
- Example: `def add : Nat → Nat → Nat`

Theorem (`theorem`):

- States a property (*proposition*)
- Provides a proof
- Example: `theorem add_comm : add m n = add n m`

Key point: Both are functions in Lean's type theory!

Axioms and Opaque Definitions

Axioms: Theorems without proofs (dangerous!)

```
opaque a : ℤ
opaque b : ℤ

axiom a_less_b : a < b  -- Assumed without proof

-- Can use in other proofs
theorem a_not_equal_b : a ≠ b := by
  intro h
  have : a < a := h ? a_less_b
  omega  -- Contradiction!
```

Warning: Axioms can introduce inconsistencies!

- `axiom false_axiom : False` breaks everything
- Use `axiom` only when absolutely necessary

Standard Axioms in Lean

Lean includes some standard axioms:

1. **Propositional extensionality**:
   - Two propositions are equal if they're logically equivalent
   - `(P ↔ Q) → (P = Q)`
2. **Quotient types**:
   - Construct types from equivalence relations
   - Needed for mathematical structures
3. **Classical logic**:
   - Law of excluded middle: `P ∨ ¬P`
   - Choice: `(∀ x, ∃ y, P x y) → ∃ f, ∀ x, P x (f x)`

Section 9

Dependent Types

What are Dependent Types?

Simple types: Types don't depend on values

- `List Nat`: type doesn't know list length
- `Nat → Nat`: function type doesn't specify behavior
- `Array Int`: array size not in type

Dependent types: Types CAN depend on values!

- `Vector Nat 3`: vector of exactly 3 natural numbers
- `(n : Nat) → Vector Nat n`: returns vector of length n
- `{i : Nat // i < 10}`: natural numbers less than 10

Why powerful? Encode properties in types!

Dependent Types Example: Vectors

Problem: Lists don't track length in type

```
-- Regular list (length unknown)
def badHead (xs : List Nat) : Nat :=
  xs.head!  -- Crashes if empty

-- Dependent type solution: Vector (list with length)
inductive Vector (α : Type) : Nat → Type where
  | nil  : Vector α 0                              -- Empty vector
  | cons : α → {n : Nat} → Vector α n → Vector α (n + 1)

-- Now head is SAFE - type guarantees non-empty!
def head {α : Type} {n : Nat} (v : Vector α (n + 1)) : α :=
  match v with
  | Vector.cons x _ ⇒ x  -- No such crashes possible
```

Key: Type `Vector α n` depends on value `n : Nat`!

Vector Operations

Operations that preserve length:

Append:

- Type: `Vector α n → Vector α m → Vector α (n + m)`
- Result length is the sum of inputs!

Map:

- Type: `(α → β) → Vector α n → Vector β n`
- Preserves the length exactly!

Zip:

- Type: `Vector α n → Vector β n → Vector (α × β) n`
- Requires the same length (enforced by types!)

Dependent Types Example: Bounded Numbers

Fin n: Natural numbers less than n

```
-- Fin n = {i : Nat // i < n}
#check Fin 10            -- Type (numbers 0-9)
#check (5 : Fin 10)      -- Valid: 5 < 10
#check (15 : Fin 10)     -- (wrong) Type error: 15 not < 10

-- Safe array indexing
def safeGet {α : Type} {n : Nat} (arr : Array α)
  (h : arr.size = n) (i : Fin n) : α :=
  arr[i]  -- No bounds check needed -> type guarantees!

-- Example usage
def myArray : Array Nat := #[10, 20, 30, 40, 50]

#eval safeGet myArray rfl (2 : Fin 5)  -- Result: 30 (safe!)
```

Benefit: Eliminates array bounds exceptions at compile time!

Subtype: Refined Types

Subtype: Type with a predicate

Syntax: `{x : α // P x}`

Read: "Values of type α such that $P(x)$ holds"

Examples:

- `{n : Nat // n < 10}`: natural numbers less than 10
- `{xs : List Nat // xs.Sorted}`: sorted lists
- `{x : Int // x ≠ 0}`: non-zero integers

Construction: Provide value + proof

- `⟨5, proof_that_5_lt_10⟩ : {n : Nat // n < 10}`

Dependent Function Types (Pi Types)

Pi types (Π-types): Functions where output type depends on input value

```
-- Type depends on the value n
def pick (n : ℕ) : {i : ℕ // i ≤ n} :=
  ⟨n, Nat.le_refl n⟩  -- Return n with proof n ≤ n

#check pick           -- (n : ℕ) → {i : ℕ // i ≤ n}
#eval (pick 5).val    -- Result: 5

-- Polymorphic identity has dependent type
def id {α : Type} (x : α) : α := x
#check @id  -- {α : Type} → α → α
-- Read: "For any type α, given x : α, return something of type α"
```

Key: The output type depends on the input!

To be clear: They generalize standard functions by allowing the result type to be a 'dynamic' calculation based on the specific input value, e.g. a function that returns a proof specifically tailored to the number provided as an argument.

More Pi Type Examples

Consider the `replicate` function that creates a list of n copies of x implemented via Π-types.

```
-- Replicate: Creates a list of n copies of x
def replicate {α : Type} (n : Nat) (x : α) : List α :=
  match n with
  | 0 ⇒ []
  | n+1 ⇒ x :: replicate n x

#eval replicate 5 "hi"  -- Result: ["hi", "hi", "hi", "hi", "hi"]

-- The type of replicate: {α : Type} → (n : Nat) → α → List α
-- The list type doesn't depend on n, but it could...

-- For instance, it could return a vector:
-- {α : Type} → (n : Nat) → α → Vector α n
```

While the `replicate` function shown here returns a standard **List**, Π-types allow us to be even more precise by returning a 'Vector' $\prod_{n:\mathbb{N}} \mathbf{Vec}(\mathbb{R}, n)$, a data type that **encodes** its own length directly **into its type signature for compile-time safety**.

Dependent Types in Practice

Real-world use cases:

1. Proven-correct sorting:

```
-- Return value is proven to be sorted!
def sort (xs : List Nat) :
  {ys : List Nat // ys.Sorted ∧ ys.length = xs.length} :=
  sorry  -- (implementation with proof here)
```

2. Matrix multiplication with dimension checking:

```
def matmul {m n p : Nat}
  (A : Matrix m n) (B : Matrix n p) : Matrix m p :=
  sorry  -- (the type ensures the dimensions match)
```

Protocol State Machines

3. Protocol state machines:

```
inductive ConnectionState where
  | disconnected : ConnectionState
  | connected : ConnectionState

-- Connection type DEPENDS on state
def Connection : ConnectionState → Type :=
  fun state ⇒ match state with
    | .disconnected ⇒ Unit
    | .connected ⇒ { handle : Nat }

-- Can only send after it connected, this is directly enforced by the type!
def send (conn : Connection .connected) (msg : String) : IO Unit :=
  sorry
```

Benefit: Protocol violations become type errors!

Dependent Types: Term Depending on...

Term depending on a term:

```
fun x : Nat ⇒ x + 1
```

- Regular function:
  input (term) → output (term)
- This is just ordinary simple type theory

Term depending on a type:

$$\texttt{fun } \{\alpha \texttt{ : Type}\} \texttt{ (x : } \alpha\texttt{) ⇒ x}$$

- Polymorphic function: takes type α,
  then term $x : \alpha$
- This is *parametric polymorphism*

Type depending on a type:

$$\texttt{List : Type} \to \texttt{Type}$$

- Type constructor: takes type α,
  returns type $\texttt{List}\ \alpha$
- This is a *type-level function*

These are the building blocks of the Lambda Cube.

Dependent Types: Type Depending on Term

The key case – Type depending on a term:

$$\texttt{Vector}\ \alpha\ :\ \texttt{Nat} \rightarrow \texttt{Type}$$

- Takes a *value* n : `Nat` (a term!)
- Returns a *type* `Vector` α n
- Different values give different types!

This is what "dependent type" strictly means:

- A type family indexed by values
- The type depends on runtime data
- This would not have been possible in simple type theory

More examples:

- `Fin : Nat` $\rightarrow$ `Type` – numbers less than n
- `Matrix m n : Nat` $\rightarrow$ `Nat` $\rightarrow$ `Type` – $m \times n$ matrices
- `Array` α `n : Nat` $\rightarrow$ `Type` – arrays of size n

This property allows Lean to verify array bounds at compile-time.

Barendregt's λ-cube (1)

In the **Calculus of Constructions** (Lean's foundation):

Body (t)		Argument (x)	Description
Term	depending on	Term	Simply typed function `fun x : Nat ⇒ x + 1`
Type	depending on	Term	Dependent type `Vector α : Nat → Type`
Term	depending on	Type	Polymorphic term `fun {α} (x : α) ⇒ x`
Type	depending on	Type	Type constructor `List : Type → Type`

Lean supports *all four corners* of the λ-cube!

Barendregt's λ-cube (2)

Type systems hierarchy ("$+$" = "adds the ability to form..." / "introduces dependency..."):

- **Simply typed λ-calculus**: Term → Term
- **System F** (polymorphism): + Term → Type
- **System F_ω** (type operators): + Type → Type
- λP (dependent types): + Type → Term
- **Calculus of Constructions**: All four!

Each corner adds "expressiveness":

- Polymorphism – Generic functions
- Type operators – Generic type constructors
- Dependent types – Types that depend on values

Section 10

The Lean Architecture

Review: From Source Code to the Verified Kernel

Lean's architecture (Review):
Trusted kernel with an untrusted elaborator

1. **Source code** (what you write):
   - High-level, readable syntax
   - Type inference, implicit arguments
   - Tactics, notation, macros

2. **Elaborator** (untrusted):
   - Fills in implicit arguments
   - Resolves type class instances
   - Expands macros and notation
   - Compiles tactics to proof terms

3. **Kernel** (trusted):
   - Small, verified core ($\sim$10k lines)
   - Type checks all terms
   - Ensures logical soundness
   - De Bruijn indices, no names

For Culture: The De Bruijn Criterion

De Bruijn criterion: Trust only a small, verified kernel

Named after: Nicolaas Govert de Bruijn (1918-2012)

- Dutch mathematician
- Created Automath (first proof assistant)
- Emphasized minimal trusted base

The principle:

- Keep the trusted core as small as possible
- All proof terms flow through the kernel
- Bugs in elaborator don't compromise soundness

Key: Why This Separation Matters

Benefits:

- Elaborator can be complex without risking soundness
- Bugs in tactics don't compromise proofs
- Easy to add new features (tactics, notation)
- Kernel is small enough to verify by hand

Example: The `simp` tactic

- Elaborator: Complex rewrite engine (thousands of lines)
- Output: Simple chain of rewrite proof terms
- Kernel: Verifies each rewrite is valid
- If `simp` has a bug, kernel rejects the proof!

Motto: "Don't trust, verify" - Even if elaborator is buggy, kernel catches it!

For Culture: De Bruijn Indices

Problem with variable names: Capture, shadowing, alpha-equivalence

Solution: Use numbers instead of names!

De Bruijn index: Number of binders between variable and its binder

Examples:

- $\lambda x.\, x$ becomes $\lambda.\, 0$ (refers to nearest binder)
- $\lambda x.\, \lambda y.\, x$ becomes $\lambda.\, \lambda.\, 1$ (skip one binder)
- $\lambda x.\, \lambda y.\, y$ becomes $\lambda.\, \lambda.\, 0$ (nearest binder)

Benefit: No alpha-conversion needed! Names don't matter.

The Elaboration Pipeline

From source to kernel:

1. **Parsing**:
   - Source code → Abstract syntax tree
   - Handle notation, macros, syntax sugar
2. **Elaboration**:
   - Fill in implicit arguments
   - Resolve type class instances
   - Compile tactics to proof terms
   - Insert coercions
3. **Kernel checking**:
   - Type check elaborated term
   - Verify termination
   - Ensure soundness

Section 11

Summary

Week 2 Summary

- **Lambda Calculus** provides the computational foundation
  - Functions are first-class
  - Beta reduction for computation
  - Church encodings show universality
- **Simply Typed Lambda Calculus** adds safety
  - Type system prevents errors
  - Curry-Howard: Types = Propositions
- **Dependent Types** let types talk about values
  - Encode properties in types
  - Eliminate runtime checks
  - Express precise specifications
- **Lean's Architecture**: Trusted kernel + Flexible elaborator
  - Small trusted core (De Bruijn criterion)
  - Complex features without compromising soundness

Section 12

Assignments & Next Steps

This Week's Assignments

Readings (see the course website)

- Theorem Proving in Lean 4 – Chapters 2-3
- The Hitchhiker's Guide to Logical Verification – Chapters 1-2
- Functional Programming in Lean 4 – Chapter 1

"Hand-in" Assignments (see the course website)

- PROOF101 Quiz 2 (due next time)
- **Programming Assignment 2: Lambda Calculus & Type Theory (due next time)**

Questions & Discussion

Questions?

Join our community:

Discord: Link on website

WhatsApp: Link on website

Website: https://danieldia-dev.github.io/proofs/

Email: dmd13@mail.aub.edu

"In mathematics, you don't understand things. You just get used to them."

— John von Neumann

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society

Spring 2026

Week 2 of 10

Dependent Type Theory

Daniel Dia & Guest Lecturers

`https://danieldia-dev.github.io/proofs/`