Dependent Type Theory for Absolute Beginners

Created with the help of GPT-5 Pro

Kimon Fountoulakis

October 14, 2025

Contents

1 Types and universes

Types

In dependent type theory, all objects we manipulate are *terms*, and each term has a *type*. For example, a natural number 0 has type Nat, and a boolean value true has type Bool. We use the colon notation $a : A$ to mean that a is a term of the type A. For example:

$$0 : \mathsf{Nat}, \quad \mathsf{false} : \mathsf{Bool}, \quad \text{“}\textit{hello world}\text{”} : \mathsf{String}.$$

Universes

But what about Nat and Bool themselves? They too are objects in the theory and therefore must have a type. The type to which all ordinary types belong is called the *universe of types* and is written Type. Thus we write

$$\mathsf{Nat} : \mathsf{Type}, \qquad \mathsf{Bool} : \mathsf{Type}, \qquad \mathsf{String} : \mathsf{Type}.$$

Intuitively, Type is “the type of all types.” However, if we were to write $\mathsf{Type} : \mathsf{Type}$, we would obtain an inconsistency (Girard’s paradox). To avoid this, type theory introduces an infinite hierarchy of universes:

$$\mathsf{Type}_0,\ \mathsf{Type}_1,\ \mathsf{Type}_2,\ \ldots$$

Each universe is itself a type in the next one, forming a cumulative hierarchy:

$$\mathsf{Type}_m : \mathsf{Type}_n \quad \text{for } m < n.$$

Intuitively, Type_0 is the universe of “small” types (such as Nat, Bool, etc.), Type_1 is the universe of “types of small types,” and so on. “In practice,” we often omit universe indices and simply write Type when the specific level is irrelevant.

2 Introduction to the notation $\lambda x : A.\, t$

The expression

$$\lambda x : A.\, t$$

is called a *lambda abstraction* or *function definition*. It denotes a function that takes an input x of type A and returns the term t as its output.

Structure

The components of the expression are as follows:

λ	(the “lambda” symbol introducing a function);
x	(the variable name of the function argument);
$: A$	(the type annotation for the variable);
$.$	(a separator between the argument and the function body);
t	(the function body, possibly depending on x).

Hence, the expression

$$\lambda x : A.\, t$$

should be read as

"the function that takes an argument x of type A and returns t."

Examples

Identity function.

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

This denotes the function that takes a natural number x and returns x itself. Its type is $\mathsf{Nat} \to \mathsf{Nat}$.

Constant function.

$$\lambda x : \mathsf{Nat}.\, 0$$

This function ignores its input and always returns 0. Type: $\mathsf{Nat} \to \mathsf{Nat}$.

Boolean negation.

$$\lambda b : \mathsf{Bool}.\ \mathsf{case}(b,\ \lambda\_.\, \mathsf{false},\ \lambda\_.\, \mathsf{true})$$

This represents the Boolean negation function, of type $\mathsf{Bool} \to \mathsf{Bool}$. The expression

$$\mathsf{case}(b,\ \lambda\_.\, \mathsf{false},\ \lambda\_.\, \mathsf{true})$$

is the *eliminator* (or destructor) for the boolean type. It performs a case distinction depending on whether b is true or false:

$$\mathsf{case}(b,\ \lambda\_.\, t_1,\ \lambda\_.\, t_2) \text{ means } \begin{cases} t_1 & \text{if } b = \mathsf{true}, \\ t_2 & \text{if } b = \mathsf{false}. \end{cases}$$

The underscore "_" denotes an *ignored argument*; for instance, $\lambda\_.\, \mathsf{false}$ is the function that always returns false regardless of its input. Hence

$$\lambda b : \mathsf{Bool}.\ \mathsf{case}(b,\ \lambda\_.\, \mathsf{false},\ \lambda\_.\, \mathsf{true})$$

defines the boolean negation function, i.e. a function that returns false when b is true and true when b is false.

Function application and computation

If f is a lambda abstraction and a is an argument of the appropriate type, we apply $f \equiv \lambda x : A.\, t$ to a by writing

$$f\, a.$$

The result is obtained by substituting a for all free occurrences of x in the body t:

$$(\lambda x\!:\!A.\, t)\, a \;\equiv\; t[a/x].$$

This is called the *beta reduction* (β-reduction), and it formalizes the usual idea of function application.

Example.

$$(\lambda x\!:\!\mathsf{Nat}.\, x + 1)\, 3 \;\equiv\; 3 + 1 \;\equiv\; 4.$$

Intuitively, the lambda abstraction defines a function, and applying it "plugs in" the argument.

Relation to ordinary mathematics

The notation

$$\lambda x\!:\!A.\, t$$

corresponds to the mathematical function expression

$$x \mapsto t(x).$$

The lambda notation, however, treats functions as *first-class terms*—they can be passed as arguments, returned as results, and manipulated like any other expression.

3 Contexts

The empty context

The symbol $\cdot$, often written simply as $\cdot$, denotes the *empty context*. It represents the starting point of all contexts—a situation with no assumptions.

General contexts

First, some terminology, $x : A$ is a called a *variable declaration*. A context is simply a finite list of typed variable declarations. For example:

$$\cdot, \qquad x : \mathsf{Nat}, \qquad x : \mathsf{Nat},\, y : \mathsf{Bool}, \qquad x : \mathsf{Nat},\, y : \mathsf{Bool},\, z : \mathsf{String}.$$

These are four different contexts. The first, i.e., $\cdot$, denotes the empty context. The second includes only one declaration, the third includes two declarations and the forth includes three declarations. Each larger context is obtained from a smaller one by adding one more declaration.

Extension of a context

Let Γ be a context, then $\Gamma, x : A$ is the *extension* of Γ with a new declaration.

Example of construction.

$$\Gamma_0 = \cdot, \qquad \Gamma_1 = \Gamma_0, x : \mathsf{Nat}, \qquad \Gamma_2 = \Gamma_1, y : \mathsf{Bool}, \qquad \Gamma_3 = \Gamma_2, z : \mathsf{String}.$$

Hence

$$\Gamma_3 = x : \mathsf{Nat},\ y : \mathsf{Bool},\ z : \mathsf{String}.$$

This shows how a recursive definition generates all finite contexts.

Inductive definition of contexts

Formally, we define the *syntax of contexts* using the notation

$$\Gamma ::= \cdot \mid \Gamma,\ x : A.$$

The symbol `::=` is read as "is defined as," and the vertical bar "|" means "or." Thus, this definition should be read as:

> A context Γ is either the empty context $\cdot$, or an existing context extended by a new variable declaration $x : A$.

This is a *recursive* or *inductive* definition:

- The *base case* says that the empty context $\cdot$ is a valid context.
- The *inductive case* says that if Γ is a valid context and A is a type, then $\Gamma, x : A$ is also a valid context.

The domain of a context Γ is denoted by $\mathsf{dom}(\Gamma)$, and it is the set of variables declared in Γ.

4 Judgments

A *judgment* is a basic assertion of the form we can derive in the theory.

We will use the following judgment forms throughout:

> $\Gamma \vdash t : A$, reads "under context Γ, the term t has type A.

and

> $\Gamma \vdash a \equiv b : A$, reads "under context Γ, the terms a and b are judgementally equal at type A,

and

> $\Gamma \vdash \mathsf{ctx}$, reads "context Γ is well-formed.

Judgements are derived using inference rules, which we will discuss later. For now, let's explain judgementally equivalent means and what a well-formed context means.

Definition (judgmental/definitional equality).

The judgment $\Gamma \vdash a \equiv b : A$ means that a and b are *judgmentally equal at type* A. The equallity holds by computation and definitional unfolding.

Example. Let

$$f \;:\equiv\; \lambda x{:}\mathsf{Nat}.\, x + 1.$$

Then by definition,

$$f\,2 \;\equiv\; (\lambda x{:}\mathsf{Nat}.\, x + 1)\,2 \;\equiv\; 2 + 1 \;\equiv\; 3.$$

Hence we can write the judgment

$$\cdot \vdash f\,2 \equiv 3 : \mathsf{Nat},$$

which reads: "in the empty context, $f\,2$ and 3 are judgmentally equal at type Nat."

Well-formed contexts ($\Gamma \vdash \mathsf{ctx}$)

We write the *well-formedness* judgment

$$\Gamma \vdash \mathsf{ctx}$$

and read it as "Γ is a well-formed context." The symbol ctx is a fixed tag (a nullary predicate) used only on the right of $\vdash$ to denote this judgment; it is not a type.

What does it mean for a context to be well-formed? Intuitively, a context Γ is *well-formed* if every declaration inside it makes sense: each type appearing in a declaration is itself already a valid type in the smaller context preceding it. Formally, recall that a context is a sequence of variable declarations:

$$\Gamma :\equiv x_1 : A_1,\, x_2 : A_2,\, \ldots,\, x_n : A_n.$$

We say that such a context is *well-formed*, written

$$\Gamma \vdash \mathsf{ctx},$$

if and only if the following recursive conditions hold:

- The empty context is well-formed:

$$\cdot \vdash \mathsf{ctx}.$$

- If $\Gamma \vdash \mathsf{ctx}$ and the type A is well-formed under Γ, i.e. $\Gamma \vdash A : \mathsf{Type}$, and the variable x is fresh ($x \notin \mathsf{dom}(\Gamma)$), then the extended context $\Gamma, x : A$ is well-formed:

$$\Gamma, x : A \vdash \mathsf{ctx}.$$

Intuitively, this means that the declarations in a context must be arranged so that each type depends only on variables that have been declared earlier.

Example.

$$x : \mathsf{Nat},\ y : \mathsf{Bool},\ z : \mathsf{String} \vdash \mathsf{ctx}$$

is well-formed, since each type (Nat, Bool, String) is already a valid type in the previous context.

Dependent example.

$$x : \mathsf{Nat},\ y : (\mathsf{Bool}, x) \vdash \mathsf{ctx}$$

is also well-formed, because (Bool, x) is a type depending on x, and x has already been declared.

Non-example.

$$y : (\mathsf{Bool}, x),\ x : \mathsf{Nat} \nvdash \mathsf{ctx},$$

because y's type refers to x, but x has not yet been declared at that point.

In summary. A context is well-formed precisely when every variable declaration in it is meaningful in the smaller context built from the declarations before it. This ensures that type dependencies are acyclic and well-scoped.

5 Reading inference rules and the rule bar

An *inference rule* has the schematic form

$$\frac{\mathit{premise}_1 \quad \cdots \quad \mathit{premise}_n}{\mathit{conclusion}}\ \textsc{Name}$$

The long *horizontal rule bar* separates premises (above) from the conclusion (below). If there are no premises, the rule is an *axiom* (always available). The inference rule is used to derive judgements.

Derivation of the empty context

Empty context.

$$\frac{}{\cdot \vdash \mathsf{ctx}}\ \textsc{Ctx-Empty}$$

Derivation of an extension

First, we provide one useful definition.

Domain and freshness. The set of variables declared in Γ is its *domain*, written $\mathsf{dom}(\Gamma)$. It is defined inductively by

$$\mathsf{dom}(\cdot) :\equiv \emptyset, \qquad \mathsf{dom}(\Gamma, x : A) :\equiv \mathsf{dom}(\Gamma) \cup \{x\}.$$

We write $x \notin \mathsf{dom}(\Gamma)$ to express that x is *fresh* for Γ. We use the usual membership notation $(x : A) \in \Gamma$ to mean that the declaration $x : A$ occurs somewhere in Γ.

Extension.

$$\frac{\Gamma \vdash \mathsf{ctx} \quad \Gamma \vdash A : \mathsf{Type} \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x : A \vdash \mathsf{ctx}}\ \textsc{Ctx-Ext}$$

The side condition $x \notin \mathsf{dom}(\Gamma)$ enforces that variable names in a context are pairwise distinct. Sometimes, the side condition is not mentioned explicitly as a premise.

Derivation of variables

From $\Gamma \vdash \mathsf{ctx}$ and $(x : A) \in \Gamma$ we may derive the trivial *variable rule*:

$$\frac{\Gamma \vdash \mathsf{ctx} \quad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

Derivation of judgements

A *derivation* of a judgment is a tree built from inference rules, with the judgment we want to justify placed at the root. Each node of the tree is an application of a rule whose premises appear above it and whose conclusion appears below the horizontal line. For example, using the rules defined so far, we can derive the judgment

$$\cdot \vdash x : \mathbf{1},$$

where $\mathbf{1}$ is the *unit type*. The unit type, written $\mathbf{1} : \mathsf{Type}_0$, is a type with exactly one term, commonly written $\star$:

$$\mathbf{1} : \mathsf{Type}_0, \qquad \star : \mathbf{1}.$$

The full derivation tree is:

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash \mathsf{ctx}}\ \textsc{Ctx-Empty} \qquad \dfrac{}{\cdot \vdash \mathbf{1} : \mathsf{Type}_0}\ \mathbf{1}\textsc{-Form} \qquad x \notin \mathsf{dom}(\cdot)}{x : \mathbf{1} \vdash \mathsf{ctx}}\ \textsc{Ctx-Ext} \qquad (x : \mathbf{1}) \in (x : \mathbf{1})}{x : \mathbf{1} \vdash x : \mathbf{1}}\ \textsc{Var}$$

This tree reads bottom-up as follows:

- By Ctx-Empty, the empty context is well-formed.
- By $\mathbf{1}$-Form, the unit type $\mathbf{1}$ is a type in the lowest universe Type_0.

- By Ctx-Ext, extending the empty context with $x : \mathbf{1}$ gives a well-formed context $x : \mathbf{1} \vdash \mathsf{ctx}$.
- By Var, from $x : \mathbf{1} \vdash \mathsf{ctx}$ we can derive $x : \mathbf{1} \vdash x : \mathbf{1}$.

Hence the complete derivation establishes

$$\cdot \vdash x : \mathbf{1}.$$

Derivation of well-formedness

We illustrate how to apply the context rules

$$\frac{}{\cdot \vdash \mathsf{ctx}}\ \textsc{Ctx-Empty} \qquad \frac{\Gamma \vdash \mathsf{ctx} \quad \Gamma \vdash A : \mathsf{Type} \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x{:}A \vdash \mathsf{ctx}}\ \textsc{Ctx-Ext}$$

to derive a well-formed context.

Comment. Often, $\Gamma \vdash \mathsf{ctx}$ is not added as a premise, because we already assume $\Gamma \vdash A : \mathsf{Type}$ in the premise list; and for this premise to be valid, it must already be the case that $\Gamma \vdash \mathsf{ctx}$.

Let's now proceed to a few examples. Throughout, assume the base types are available in the empty context:

$$\cdot \vdash \mathsf{Nat} : \mathsf{Type}, \qquad \cdot \vdash \mathsf{Bool} : \mathsf{Type}, \qquad \cdot \vdash \mathsf{String} : \mathsf{Type}.$$

Example 1 (non-dependent). Check $x : \mathsf{Nat}, y : \mathsf{Bool} \vdash \mathsf{ctx}$.

$$\dfrac{\dfrac{\cdot \vdash \mathsf{ctx}\ \textsc{Ctx-Empty} \qquad \cdot \vdash \mathsf{Nat} : \mathsf{Type} \qquad x \notin \mathsf{dom}(\cdot)}{x : \mathsf{Nat} \vdash \mathsf{ctx}}\ \textsc{Ctx-Ext} \qquad x : \mathsf{Nat} \vdash \mathsf{Bool} : \mathsf{Type} \qquad y \notin \mathsf{dom}(x : \mathsf{Nat})}{x : \mathsf{Nat}, y : \mathsf{Bool} \vdash \mathsf{ctx}}\ \textsc{Ctx-Ext}$$

Reading bottom-up: start with $\cdot$ via Ctx-Empty; extend by $x : \mathsf{Nat}$ using $\cdot \vdash \mathsf{Nat} : \mathsf{Type}$; then extend by $y : \mathsf{Bool}$ using $x : \mathsf{Nat} \vdash \mathsf{Bool} : \mathsf{Type}$ and freshness.

Non-example (dependency out of order). Consider $y : (\mathsf{Bool}, x)$, $x : \mathsf{Nat}$. Attempting Ctx-Ext on the first declaration requires $\cdot \vdash (\mathsf{Bool}, x) : \mathsf{Type}$. But in $\cdot$ there is no variable $x : \mathsf{Nat}$, so we cannot derive $\cdot \vdash x : \mathsf{Nat}$, hence $\cdot \vdash (\mathsf{Bool}, x) : \mathsf{Type}$ fails. Therefore

$$y : (\mathsf{Bool}, x),\ x : \mathsf{Nat} \nvdash \mathsf{ctx}.$$

Non-example (duplicate name). Consider $x : \mathsf{Nat}, x : \mathsf{Bool}$. The second extension violates freshness since $x \in \mathsf{dom}(x : \mathsf{Nat})$. Thus the side condition $x \notin \mathsf{dom}(\Gamma)$ fails and

$$x : \mathsf{Nat},\ x : \mathsf{Bool} \nvdash \mathsf{ctx}.$$

These derivations show that checking $\Gamma \vdash \mathsf{ctx}$ reduces *recursively* to (i) checking the smaller prefix is well-formed, (ii) checking the new declaration's type is a type in that prefix, and (iii) enforcing freshness.

6 Structural rules and judgmental equality

Capture-avoiding substitution

When we write a substitution $t[a/x]$, we mean the process of *replacing all free occurrences* of the variable x in the term t by the term a. However, care must be taken when t contains *binders* (such as $\lambda y.\, u$ or a context declaration $y : B$) that introduce new variables.

The problem: variable capture. If we substitute naively, a free variable of a might become *accidentally bound* by one of these binders. This error is called *variable capture*.

Example (the bad case). Consider:

$$t \;=\; \lambda y.\, x + y, \qquad a \;=\; y.$$

A naive substitution $t[a/x]$ would yield

$$\lambda y.\, y + y,$$

but now the free y from a has been captured by the binder $\lambda y.$, changing its meaning completely. Originally, the inner y in a referred to some outer variable, but after substitution it refers to the bound parameter of the lambda.

The solution: capture-avoidance. Before performing substitution, we *rename bound variables* in t so that they do not clash with the free variables of a. This is called *capture-avoiding substitution*.

In the example above, we first rename the bound variable y in t to a fresh variable y':

$$t' \;:\equiv\; \lambda y'.\, x + y'.$$

Now we can safely substitute a for x:

$$t'[a/x] \;=\; \lambda y'.\, y + y'.$$

No variable has been captured, and the meaning is preserved.

Summary.

- *Variable capture* occurs when a free variable in the substituting term becomes bound by a binder in the target expression.
- *Capture-avoiding substitution* prevents this by systematically renaming bound variables before substitution.
- This ensures that substitution preserves the intended meaning of terms.

Preliminaries: explanation of Δ notation and substitution brackets $[\,\cdot\,/\,\cdot\,]$

In what follows we write contexts of the form

$$\Gamma,\, x : A,\, \Delta.$$

Here:

- Γ is the initial prefix of the context.
- $x : A$ is the current variable declaration we are focusing on.
- Δ denotes the *remainder of the context* after $x : A$. It may contain additional declarations types that can depend on x. Formally, if

  $$\Delta \;:\equiv\; y_1 : B_1,\, y_2 : B_2,\, \ldots,\, y_k : B_k,$$

  then the complete context is

  $$\Gamma,\, x : A,\, y_1 : B_1,\, y_2 : B_2,\, \ldots,\, y_k : B_k.$$

Substitution brackets. The notation $t[a/x]$ means *capture-avoiding substitution* of the term a for the variable x in the expression t. It replaces all free occurrences of x in t by a, renaming bound variables when necessary to avoid name capture.

Similarly, for contexts we write $\Delta[a/x]$ to mean "substitute a for x in every type declared in Δ". If

$$\Delta \;:\equiv\; y_1 : B_1,\, y_2 : B_2,\, \ldots,\, y_k : B_k,$$

then

$$\Delta[a/x] \;:\equiv\; y_1 : B_1[a/x],\, y_2 : B_2[a/x],\, \ldots,\, y_k : B_k[a/x].$$

Properties.

- If x does not occur free in Δ, then $\Delta[a/x] = \Delta$.
- Substitution respects binding and avoids variable capture (by α-conversion if necessary).

If $\Gamma, x : A, \Delta \vdash b : B$, then $b[a/x]$ denotes the term obtained by substituting a for x in b.

Thus, in the substitution rules, Δ represents the *tail of the context*, and the square brackets $[a/x]$ represent standard, capture-avoiding substitution applied to terms, types, or all declarations within Δ.

Variable, substitution, and weakening

The following *structural* principles are admissible (provable by induction on derivations) and may be used freely.

Variable. From a well-formed context, any declared variable has its declared type.

$$\frac{\Gamma \vdash \mathsf{ctx} \quad (x{:}A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

Substitution (typing). If $a : A$ is derivable in Γ and, under an extended context $\Gamma, x{:}A, \Delta$, a judgment $\cdot \vdash b : B$ is derivable, then we may substitute a for x.

$$\frac{\Gamma \vdash a : A \quad \Gamma, x{:}A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \ : \ B[a/x]}\ \textsc{Subst}_1$$

Weakening (typing). If A is a type in Γ and some judgment holds in Γ, Δ, we may insert a fresh, unused declaration $x{:}A$ anywhere between Γ and Δ.

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, \Delta \vdash b : B}{\Gamma, x{:}A, \Delta \vdash b : B}\ \textsc{Wkg}_1$$

Substitution (judgmental equality).

$$\frac{\Gamma \vdash a : A \quad \Gamma, x{:}A, \Delta \vdash b \equiv c : B}{\Gamma, \Delta[a/x] \vdash b[a/x] \ \equiv \ c[a/x] \ : \ B[a/x]}\ \textsc{Subst}_2$$

Weakening (judgmental equality).

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, \Delta \vdash b \equiv c : B}{\Gamma, x{:}A, \Delta \vdash b \equiv c : B}\ \textsc{Wkg}_2$$

As usual, side conditions ensure $x \notin \mathsf{dom}(\Gamma)$ when extending a context.

Judgmental equality: laws and conversion

We assume judgmental equality $\Gamma \vdash a \equiv b : A$ is an *equivalence relation* and is *respected by typing*. Concretely, we use the following admissible rules.

Equivalence laws (at a fixed type).

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A}\ \textsc{Refl} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A}\ \textsc{Sym} \qquad \frac{\Gamma \vdash a \equiv b : A \ \ \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}\ \textsc{Trans}$$

Conversion (type equality transports typing).

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \equiv B : \mathsf{Type}_i}{\Gamma \vdash a : B}\ \textsc{Conv-Ty} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash A \equiv B : \mathsf{Type}_i}{\Gamma \vdash a \equiv b : B}\ \textsc{Conv-Eq}$$

7 Type universes (re-visited)

We postulate an infinite hierarchy of universes of types:

$$\mathsf{Type}_0, \quad \mathsf{Type}_1, \quad \mathsf{Type}_2, \quad \ldots$$

Each universe is contained in the next one, and any type in Type_i is also a type in Type_{i+1}. Formally, we have the following rules:

$$\frac{\Gamma \vdash \mathsf{ctx}}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ \mathsf{Type}\text{-}\textsc{Intro} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash A : \mathsf{Type}_{i+1}}\ \mathsf{Type}\text{-}\textsc{Cumul}$$

Explanation.

- The first rule (Type-Intro) states that each universe Type_i itself has a type in the next higher universe Type_{i+1}.
- The second rule (Type-Cumul) expresses *cumulativity*: if a type A belongs to some universe Type_i, it is also regarded as a type in every higher universe.

Remarks. We set up the rules of the type theory so that whenever a typing judgment $\Gamma \vdash a : A$ holds, it follows that $\Gamma \vdash A : \mathsf{Type}_i$ for some universe index i. In other words, every type A always lives in some universe Type_i.

Furthermore, judgmental equality preserves typing: if $\Gamma \vdash a \equiv b : A$ then both a and b have type A, i.e.

$$\Gamma \vdash a \equiv b : A \quad \Rightarrow \quad \Gamma \vdash a : A \text{ and } \Gamma \vdash b : A.$$

8 Rules associated with a type

Each type in dependent type theory is characterized by a collection of rules that specify how it can be formed, inhabited, used, and reasoned about:

- **Formation rule**, stating when the type former can be applied;
- **Introduction rules**, stating how to inhabit the type;
- **Elimination rules**, or an induction principle, stating how to use an element of the type;
- **Computation rules**, which are judgmental equalities explaining what happens when elimination rules are applied to results of introduction rules;
- (optional) **Uniqueness principles**, which are judgmental equalities explaining how every element of the type is uniquely determined by the results of elimination rules applied to it.

9 Functions and the arrow type ($A \to B$)

A *function* from a type A to a type B is a term of the *function type* $A \to B$. Intuitively, a function transforms any input $a : A$ into an output $b : B$. In type theory, functions are introduced by *lambda abstraction* and used by *application*.

Rules for the arrow type

We present the usual four rules: *formation*, *introduction* (lambda), *elimination* (application), and *computation* (β-reduction).

Formation. If A and B are types, then $A \to B$ is a type:

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \to B : \mathsf{Type}} \to\text{-Form}$$

Introduction (lambda abstraction). If under the assumption $x : A$ we can build a term $t : B$, then $\lambda x.\, t$ is a function $A \to B$:

$$\frac{\Gamma,\, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \to\text{-Intro}$$

We often write $\lambda x : A.\, t$ to annotate the parameter type explicitly. Note that t here represents an expression that potentially depends on x. So, strictly, speaking $\lambda x.\, t$ is just a "name" of a function, it's not an explicitly stated function.

Elimination (application). Given a function $f : A \to B$ and an argument $a : A$, we may *apply* f to a:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B} \to\text{-Elim}$$

Computation (β-reduction). Applying a lambda to an argument computes by capture-avoiding substitution:

$$\frac{\Gamma,\, x : A \vdash t : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.\, t)\, a \equiv t[a/x] : B} \to\text{-Comp-}\beta$$

Here $t[a/x]$ denotes capture-avoiding substitution of a for x in t. This rule specifies the computation *only when the function is (definitionally) a λ-abstraction*. In general, an application $f\, a$ need not reduce unless f *unfolds* to a lambda; otherwise $f\, a$ is a neutral term.

Difference between elimination and computation. It is important to distinguish the *elimination* rule from the *computation* rule.

- **Elimination rule.** The elimination rule specifies *how to use* or *consume* a term of a given type. It allows us to produce something else from a value of that type. For function types, the elimination rule is function application:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B}\ \to\text{-Elim}$$

  This rule does not specify what $f\, a$ *evaluates to*; it only states that such a term is well-typed.

- **Computation rule.** The computation rule (often called the β*-rule*) specifies *what happens* when an elimination acts on an introduction. It defines how the expression reduces or computes. For function types:

$$(\lambda x.\, t)\, a \ \equiv\ t[a/x].$$

  That is, applying a function introduced by a λ-abstraction to an argument a yields the function body with a substituted for x.

Elimination rules describe *how we may use* a term of a type, while computation rules describe *what happens when we use it*. In the case of functions, elimination is application, and computation expresses how application and λ-abstraction interact:

$$\text{Introduction } (\lambda)\ +\ \text{Elimination (application)}\ \Rightarrow\ \text{Computation } (\beta).$$

Examples

Identity and constant functions.

$$\mathsf{id}_A\ :\equiv\ \lambda x{:}A.\, x\ :\ A \to A, \qquad \mathsf{const}_{A,B}(a)\ :\equiv\ \lambda\_{:}B.\, a\ :\ B \to A.$$

The identity can be computed as $\mathsf{id}_A\, a \equiv a$.

Conditional function. We define the higher-order conditional operator

$$\mathsf{if}_A\ :\equiv\ (\lambda b{:}\mathsf{Bool}.\, (\lambda t{:}A.\, (\lambda f{:}A.\, \mathsf{case}(b,\ \lambda\_.\, t,\ \lambda\_.\, f))))\ :\ \mathsf{Bool} \to (A \to (A \to A)).$$

That is, if_A is a curried function taking three arguments: a boolean $b : \mathsf{Bool}$, and two values $t, f : A$. Its behavior is given by

$$\mathsf{if}_A\ b\ t\ f\ :\equiv\ \mathsf{case}(b,\ \lambda\_.\, t,\ \lambda\_.\, f),$$

so that

$$\mathsf{if}_A\ \mathsf{true}\ t\ f\ \equiv\ t, \qquad \mathsf{if}_A\ \mathsf{false}\ t\ f\ \equiv\ f.$$

Composition. Given $f : B \to C$ and $g : A \to B$, define

$$f \circ g \;:\equiv\; \lambda x{:}A.\, f\,(g\,x) \;:\; A \to C.$$

10 Constructors and canonical forms

In type theory, each type is defined by specifying its *constructors*—the canonical ways of producing elements (terms) of that type. Constructors determine how we can *build* terms of a given type, and by extension, how we can reason about or eliminate them. Elimination rules say how to take a value of that type apart (or act on it) to produce something else.

Formally, for a type $A : \mathsf{Type}$, a constructor is a term-forming rule of the form

$$\frac{\Gamma \vdash t_1 : A_1 \quad \ldots \quad \Gamma \vdash t_n : A_n}{\Gamma \vdash c(t_1, \ldots, t_n) : A},$$

which specifies how a new term $c(t_1, \ldots, t_n)$ of type A can be formed from existing terms of other types $A_1, \ldots, A_n$. Each type comes with one or more constructors that uniquely determine its canonical inhabitants.

Clarifying note. At this stage, the expression $c(t_1, \ldots, t_n)$ should be understood *purely syntactically*: it describes how to *form* a new term from existing ones using a constructor symbol c of arity n. The parentheses indicate syntactic application of a constructor to its arguments, not functional application. $c(t_1, \ldots, t_n)$ does not denote a function being applied to arguments, but simply the construction of a new term according to the formation rule of the type. For example, for a nullary constructor ($n = 0$) such as the unit value $\star$, the general rule specializes to

$$\frac{}{\Gamma \vdash \star : \mathbf{1}},$$

indicating that $\star$ is a canonical term of the unit type.

Syntactically, c is a *constructor symbol* for the type A, not a previously defined function you prove correct. The rule *declares* that supplying n arguments of the listed types yields a canonical element of A.

Just like for function type formation $A \to B$, this is a *rule of the calculus*: it *licenses* forming $c(t_1, \ldots, t_n)$ once the premises hold.

Other examples.

- **Empty type.** The empty type $\mathbf{0}$ has *no* constructors. Therefore, there are no canonical terms $a : \mathbf{0}$. This make sense, since by definition the empty type is empty.
- **Sum type.** For two types $A, B : \mathsf{Type}$, the sum type $A + B$ has two constructors:

$$\frac{a : A}{\mathsf{inl}(a) : A + B} \qquad \text{and} \qquad \frac{b : B}{\mathsf{inr}(b) : A + B}.$$

The first constructor inl injects values from the left component A, and the second constructor inr injects values from the right component B. As we will see later, these two rules fully describe the canonical forms of elements of $A + B$.

Intuition. Constructors are the primitive building blocks of each type. For instance, every term of $A+B$ is either constructed as $\mathsf{inl}(a)$ for some $a : A$, or as $\mathsf{inr}(b)$ for some $b : B$; there are no other canonical ways to obtain a term of this type. This property allows us to reason about sum types by *case analysis*, which we will define later.

11 Sum types and the inl and inr constructors

Given two types $A : \mathsf{Type}$ and $B : \mathsf{Type}$, their *sum type* is written

$$A + B : \mathsf{Type}.$$

But how is $A + B$ constructed? We first need to introduce the constructors of the type $A + B$. For two types $A, B : \mathsf{Type}$, the sum type $A + B$ has two constructors:

$$\frac{a : A}{\mathsf{inl}(a) : A + B} \qquad \text{and} \qquad \frac{b : B}{\mathsf{inr}(b) : A + B}.$$

The first constructor inl injects values into $A+B$ from the left component A, and the second constructor inr injects values into $A + B$ from the right component B. These two rules fully describe the canonical forms of elements of $A + B$.

Intuition. Constructors are the primitive building blocks of each type. For instance, every term of $A + B$ is either $\mathsf{inl}(a)$ for some $a : A$, or $\mathsf{inr}(b)$ for some $b : B$; there are no other canonical ways to obtain a term of this type.

Reminder. At this stage, the expression inl should be understood *purely syntactically*: it describes how to *form* a new term from existing ones using a constructor symbol inl. The parentheses indicate syntactic application of a constructor to its arguments, not functional application. We have not yet introduced functions, so inl does not denote a function being applied to arguments, but simply the construction of a new term according to the formation rule of the type.

Example: How $A+B$ looks like? To make the idea of a sum type concrete, let us take two small finite types:

$$A :\equiv \{\mathsf{red}, \mathsf{green}\}, \qquad B :\equiv \{0, 1\}.$$

Then the sum type $A + B$ consists of all elements of A tagged by inl, and all elements of B tagged by inr:

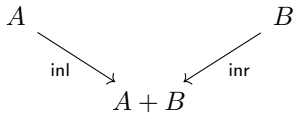
$$A + B \;=\; \{\mathsf{inl}(\mathsf{red}),\ \mathsf{inl}(\mathsf{green}),\ \mathsf{inr}(0),\ \mathsf{inr}(1)\}.$$

Intuitively, $\mathsf{inl}(a)$ means "a value coming from A, left side of the sum," and $\mathsf{inr}(b)$ means "a value coming from B, right side of the sum.".

This can be illustrated in tabular form:

Element of A	Constructor	Element of $A + B$
red	inl	$\mathsf{inl}(\mathsf{red})$
green	inl	$\mathsf{inl}(\mathsf{green})$
0	inr	$\mathsf{inr}(0)$
1	inr	$\mathsf{inr}(1)$

In general, every element of $A + B$ is either of the form $\mathsf{inl}(a)$ for some $a : A$, or $\mathsf{inr}(b)$ for some $b : B$. If desired, we can also represent the injections diagrammatically:



This expresses that both A and B "feed into" the coproduct $A + B$ using the constructors inl and inr.

Rules for the sum type $A + B$.

Formation.

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A + B : \mathsf{Type}}\ \textsc{Form}$$

Introduction.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}(a) : A + B}\ \textsc{IntroL} \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}(b) : A + B}\ \textsc{IntroR}$$

Elimination (case analysis).

$$\frac{\Gamma \vdash s : A + B \quad \Gamma \vdash f : A \to C \quad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathsf{case}(s,\ f,\ g) : C}\ \textsc{Elim}$$

$\mathsf{case}(s,\ f,\ g)$ performs a case distinction depending on whether b is true or false:

$$\mathsf{case}(s,\ f,\ g) \text{ means } \begin{cases} f & \text{if } s = \mathsf{true}, \\ g & \text{if } s = \mathsf{false}. \end{cases}$$

Computation.

$$\Gamma \vdash \mathsf{case}(\mathsf{inl}(a),\ f,\ g) \equiv f\,a : C \qquad \Gamma \vdash \mathsf{case}(\mathsf{inr}(b),\ f,\ g) \equiv g\,b : C$$

12 Product types and the pair constructor

Definition of the product type

Given two types $A : \mathsf{Type}$ and $B : \mathsf{Type}$, their *product type* is written

$$A \times B : \mathsf{Type}.$$

A term of type $A \times B$ represents a *pair* consisting of one element from A and one element from B. In other words, it corresponds to "both an A and a B together."

Constructors. The canonical way to construct a term of the product type is by using the pairing constructor:

$$\frac{a : A \quad b : B}{(a, b) : A \times B}.$$

Thus, for any $a : A$ and $b : B$, the ordered pair (a, b) is a term of $A \times B$.

Eliminators (projections). Given a term $p : A \times B$, we can *eliminate* it by projecting its components:

$$\frac{p : A \times B}{\pi_1(p) : A} \qquad \text{and} \qquad \frac{p : A \times B}{\pi_2(p) : B}.$$

Intuitively, π_1 extracts the first component and π_2 extracts the second component of the pair.

Computation rules. The product type satisfies the following definitional equalities:

$$\pi_1(a, b) \equiv a, \qquad \pi_2(a, b) \equiv b.$$

That is, projecting a pair retrieves its components directly.

Intuition. If sum types $A + B$ represent "either an A or a B," then product types $A \times B$ represent "both an A and a B." For example, if A is the type of names and B is the type of ages, then $A \times B$ is the type of *name–age pairs*.

Example. Let

$$A :\equiv \{\mathsf{red}, \mathsf{green}\}, \qquad B :\equiv \{0, 1\}.$$

Then the product type $A \times B$ consists of all possible ordered pairs:

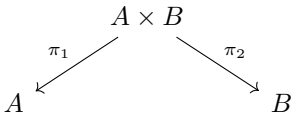
$$A \times B \;=\; \{(\mathsf{red}, 0),\ (\mathsf{red}, 1),\ (\mathsf{green}, 0),\ (\mathsf{green}, 1)\}.$$

This can be visualized in a grid form:

$\times$	0	1
red	$(\mathsf{red}, 0)$	$(\mathsf{red}, 1)$
green	$(\mathsf{green}, 0)$	$(\mathsf{green}, 1)$

Every element of $A \times B$ is one of these four pairs.

Diagrammatic intuition.



This expresses that the product type $A \times B$ can be projected back to its two components via the maps π_1 and π_2.

Rules for the non-dependent product $A \times B$

We record the standard *formation*, *introduction* (pairing), *elimination* (projections), and *computation* rules for the non-dependent product.

Formation.

$$\frac{\Gamma \vdash A : \mathsf{Type} \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}} \times\text{-Form}$$

Introduction (pairing).

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \times\text{-Intro}$$

Elimination (projections).

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_1(p) : A} \times\text{-Elim}_1 \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_2(p) : B} \times\text{-Elim}_2$$

Computation (β-rules for projections).

$$\Gamma \vdash \pi_1(a, b) \equiv a : A \qquad \Gamma \vdash \pi_2(a, b) \equiv b : B$$

These specify that projecting a canonical pair returns its components.