Dependent Type Theory for Absolute Beginners

Created with the help of GPT-5 Pro

Kimon Fountoulakis

October 8, 2025

1 Types and universes

The universe of 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:

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

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:

$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \cdots$$

Each universe Type_i is an element of the next one, but not of itself. In most practical situations, we omit universe index that a term belongs to and simply write Type.

Examples:

$$\begin{aligned} \mathsf{Nat} : \mathsf{Type}_0, &\quad \text{the type of natural numbers;} \\ \mathsf{Bool} : \mathsf{Type}_0, &\quad \text{the type of booleans;} \\ \mathsf{String} : \mathsf{Type}_0, &\quad \text{the type of strings;} \end{aligned}$$

Generally, writing $A : \mathsf{Type}$ means that A is itself a type.

The simplest types: unit and empty types

Two fundamental base types in dependent type theory are the *unit type* and the *empty type*.

The unit type. The unit type, written $\mathbf{1} : \mathsf{Type}$, is a type with exactly one canonical inhabitant, commonly written $\star$.

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

The empty type. The empty type, written $\mathbf{0} : \mathsf{Type}$, is a type with no inhabitants. There are no terms $a : \mathbf{0}$.

Product, sum and function types

More generally, if $A, B : \mathsf{Type}$, then:

$$\begin{aligned} A \times B &: \mathsf{Type} \quad \text{(product type, pairs } (a,b));\\ A + B &: \mathsf{Type} \quad \text{(sum type of } A \text{ and } B);\\ A \to B &: \mathsf{Type} \quad \text{(function type from } A \text{ to } B). \end{aligned}$$

Each of these is itself an inhabitant of the universe Type:

$$(A \to B) : \mathsf{Type}, \qquad (A \times B) : \mathsf{Type}, \qquad (A + B) : \mathsf{Type}.$$

We will provide details in subsequent sections on how these three types are constructed from types A and B.

2 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.

3 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.

Here, $\vdash$ is the *turnstile* symbol separating assumptions (on the left) from a conclusion (on the right).

Example (typing). With $\mathsf{Nat} : \mathsf{Type}$ and $\mathbf{1} : \mathsf{Type}$ (unit), the following are typing judgments:

$$\cdot \vdash \star : \mathbf{1} \qquad \text{and} \qquad x : \mathsf{Nat} \vdash x : \mathsf{Nat}.$$

The left judgement means that from the empty context we conclude that $\star$ is a term of the unit type. The right judgement is trivial. It means that if we assume that x in a natural number, then we can conclude that x is a natural number.

4 Reading inference rules and the rule bar

An *inference rule* has the schematic form

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

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

Example.

- *Unit introduction.* $$\frac{}{\Gamma \vdash \star : \mathbf{1}}\ \textsc{1-Intro}$$

The Unit introduction example reads as "without any assumptions and under any context Γ we conclude that $\star$ is of the unit type".

5 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.

Formally, for a type A : 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.

Examples.

- **Unit type.** The unit type $\mathbf{1}$ has exactly one constructor:

  $$\frac{}{\star : \mathbf{1}}.$$

  Thus, $\star$ is the only canonical inhabitant of $\mathbf{1}$. Note that this is equivalent to the "unit introduction" example. We just ommitted the context Γ, since the construction holds for any Γ.

- **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. 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.

6 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.

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.

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.

Formation rules for contexts

We inductively generate well-formed contexts with two rules.

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

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.

Basic facts

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

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

Examples and a non-example

Assume we have the closed type declarations $\cdot \vdash \mathsf{Nat} : \mathsf{Type}$, $\cdot \vdash \mathsf{Bool} : \mathsf{Type}$, and $\cdot \vdash \mathsf{String} : \mathsf{Type}$. Then the following derivations show well-formed contexts:

Example 1. $\cdot \vdash \mathsf{ctx}$ by $\textsc{Ctx-Empty}$.

Example 2.

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

Example 3.

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

Non-example (duplicate name).

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

since $x \in \mathsf{dom}(x : \mathsf{Bool})$ violates $x \notin \mathsf{dom}(x : \mathsf{Nat})$, meaning that x is not fresh.

Worked derivations: recursively checking well-formedness

We illustrate how to *recursively* 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 decide whether a given sequence of declarations forms a well-formed context. 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 $x : \mathsf{Nat},\, y : \mathsf{Bool} \vdash \mathsf{ctx}$.

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

Reading bottom-up: start with $\cdot$ via $\textsc{Ctx-Empty}$; 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.

7 Sum types and the inl and inr constructors

Definition of the sum type

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.

Comment. A potential realization of the inl and inr constructors is

$$\mathsf{inl}(a) :\equiv (\mathsf{true}, a), \quad \mathsf{inr}(b) :\equiv (\mathsf{false}, b).$$

However, they are *not* needed to use sums soundly: the inductive rules already give full computational content. This is something which will be discussed later in more depth.

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	inl(red)
green	inl	inl(green)
0	inr	inr(0)
1	inr	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:

$$\begin{tikzcd}
A \arrow[dr, "\mathsf{inl}"'] & & B \arrow[dl, "\mathsf{inr}"] \\
& A+B &
\end{tikzcd}$$

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