

Dependent Type Theory for Absolute Beginners

Created with the help of GPT-5 Pro

Kimion Fountoulakis

October 15, 2025

Contents

1	Types and universes	2
2	Introduction to the notation $\lambda x:A. t$	2
3	Contexts	4
4	Judgments	5
5	Reading inference rules and the rule bar	7
6	Structural rules and judgmental equality	10
7	Type universes (re-visited)	13
8	Rules associated with a type	13
9	Constructors and canonical forms	14
10	Non-dependent sum types and the <code>inl</code> and <code>inr</code> constructors	15
11	Non-dependent product types and the <code>pair</code> constructor	17
12	Functions and the arrow type $(A \rightarrow B)$	19
13	Dependent function types (Π-types)	21
14	Dependent pair types (Σ-types)	22

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:

$$0 : \text{Nat}, \quad \text{false} : \text{Bool}, \quad \text{"hello world"} : \text{String}.$$

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

$$\text{Nat} : \text{Type}, \quad \text{Bool} : \text{Type}, \quad \text{String} : \text{Type}.$$

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

$$\text{Type}_0, \text{Type}_1, \text{Type}_2, \dots$$

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

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

Intuitively, `Type0` is the universe of “small” types (such as `Nat`, `Bool`, etc.), `Type1` 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.

$$\lambda x:\text{Nat}. x$$

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

Constant function.

$$\lambda x:\text{Nat}. 0$$

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

Boolean negation.

$$\lambda b:\text{Bool}. \text{case}(b, \lambda_. \text{false}, \lambda_. \text{true})$$

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

$$\text{case}(b, \lambda_. \text{false}, \lambda_. \text{true})$$

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

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

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

$$\lambda b:\text{Bool}. \text{case}(b, \lambda_. \text{false}, \lambda_. \text{true})$$

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.

$$(\lambda x:\mathbf{Nat}. x + 1) 3 \equiv 3 + 1 \equiv 4.$$

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 called a *variable declaration*. A context is simply a finite list of typed variable declarations. For example:

$$\cdot, \quad x : \mathbf{Nat}, \quad x : \mathbf{Nat}, y : \mathbf{Bool}, \quad x : \mathbf{Nat}, y : \mathbf{Bool}, z : \mathbf{String}.$$

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 fourth 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.

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

Hence

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

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 “ \mid ” 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 $\text{dom}(\Gamma)$, 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

$\Gamma \vdash \text{ctx}$, 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 equality holds by computation and definitional unfolding.

Example. Let

$$f \equiv \lambda x : \text{Nat}. x + 1.$$

Then by definition,

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

Hence we can write the judgment

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

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

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

We write the *well-formedness* judgment

$$\Gamma \vdash \text{ctx}$$

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:

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

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

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

if and only if the following recursive conditions hold:

- The empty context is well-formed:

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

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

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

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.

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

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

Dependent example.

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

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

Non-example.

$$y : (\text{Bool}, x), x : \text{Nat} \not\vdash \text{ctx},$$

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

$$\frac{\text{premise}_1 \quad \cdots \quad \text{premise}_n}{\text{conclusion}} \text{NAME}$$

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.

$$\frac{}{\cdot \vdash \text{ctx}} \text{CTX-EMPTY}$$

Derivation of an extension

First, we provide one useful definition.

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

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

We write $x \notin \text{dom}(\Gamma)$ 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.

$$\frac{\Gamma \vdash \text{ctx} \quad \Gamma \vdash A : \text{Type} \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash \text{ctx}} \text{CTX-EXT}$$

The side condition $x \notin \text{dom}(\Gamma)$ 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 $\Gamma \vdash \text{ctx}$ and $(x : A) \in \Gamma$ we may derive the trivial *variable rule*:

$$\frac{\Gamma \vdash \text{ctx} \quad (x : A) \in \Gamma}{\Gamma \vdash x : A} \text{VAR}$$

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 $\mathbf{1} : \text{Type}_0$, is a type with exactly one term, commonly written \star :

$$\mathbf{1} : \text{Type}_0, \quad \star : \mathbf{1}.$$

The full derivation tree is:

$$\frac{\frac{\frac{}{\cdot \vdash \text{ctx}} \text{CTX-EMPTY} \quad \frac{}{\cdot \vdash \mathbf{1} : \text{Type}_0} \text{1-FORM}}{x : \mathbf{1} \vdash \text{ctx}} \quad \frac{x \notin \text{dom}(\cdot)}{(x : \mathbf{1}) \in (x : \mathbf{1})} \text{CTX-EXT}}{x : \mathbf{1} \vdash x : \mathbf{1}} \text{VAR}$$

This tree reads bottom-up as follows:

- By CTX-EMPTY, the empty context is well-formed.
- By 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 $x : \mathbf{1} \vdash \text{ctx}$.
- By VAR, from $x : \mathbf{1} \vdash \text{ctx}$ 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

$$\frac{}{\cdot \vdash \text{ctx}} \text{CTX-EMPTY} \quad \frac{\Gamma \vdash \text{ctx} \quad \Gamma \vdash A : \text{Type} \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash \text{ctx}} \text{CTX-EXT}$$

to derive a well-formed context.

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

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

$$\cdot \vdash \text{Nat} : \text{Type}, \quad \cdot \vdash \text{Bool} : \text{Type}, \quad \cdot \vdash \text{String} : \text{Type}.$$

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

$$\frac{\frac{\frac{\cdot \vdash \text{ctx} \text{ Ctx-EMPTY} \quad \cdot \vdash \text{Nat} : \text{Type} \quad x \notin \text{dom}(\cdot)}{x : \text{Nat} \vdash \text{ctx}} \text{ Ctx-EXT} \quad x : \text{Nat} \vdash \text{Bool} : \text{Type} \quad y \notin \text{dom}(x : \text{Nat})}{x : \text{Nat}, y : \text{Bool} \vdash \text{ctx}} \text{ Ctx-EXT}$$

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

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

$$y : (\text{Bool}, x), x : \text{Nat} \not\vdash \text{ctx}.$$

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

$$x : \text{Nat}, x : \text{Bool} \not\vdash \text{ctx}.$$

These derivations show that checking $\Gamma \vdash \text{ctx}$ 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:

$$t = \lambda y. x + y, \quad a = y.$$

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

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

then the complete context is

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

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

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

then

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

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.

$$\frac{\Gamma \vdash \text{ctx} \quad (x:A) \in \Gamma}{\Gamma \vdash x : A} \text{VAR}$$

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 .

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

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 .

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, \Delta \vdash b : B}{\Gamma, x:A, \Delta \vdash b : B} \text{WKG}_1$$

Substitution (judgmental equality).

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

Weakening (judgmental equality).

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

As usual, side conditions ensure $x \notin \text{dom}(\Gamma)$ 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).

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \text{REFL} \quad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \text{SYM} \quad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A} \text{TRANS}$$

Conversion (type equality transports typing).

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \equiv B : \text{Type}_i}{\Gamma \vdash a : B} \text{CONV-TY} \quad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash A \equiv B : \text{Type}_i}{\Gamma \vdash a \equiv b : B} \text{CONV-EQ}$$

7 Type universes (re-visited)

We postulate an infinite hierarchy of universes of types:

$$\mathbf{Type}_0, \quad \mathbf{Type}_1, \quad \mathbf{Type}_2, \quad \dots$$

Each universe is contained in the next one, and any type in \mathbf{Type}_i is also a type in \mathbf{Type}_{i+1} . Formally, we have the following rules:

$$\frac{\Gamma \vdash \text{ctx}}{\Gamma \vdash \mathbf{Type}_i : \mathbf{Type}_{i+1}} \text{Type-INTRO} \qquad \frac{\Gamma \vdash A : \mathbf{Type}_i}{\Gamma \vdash A : \mathbf{Type}_{i+1}} \text{Type-CUMUL}$$

Explanation.

- The first rule (Type-INTRO) states that each universe \mathbf{Type}_i itself has a type in the next higher universe \mathbf{Type}_{i+1} .
- The second rule (Type-CUMUL) expresses *cumulativity*: if a type A belongs to some universe \mathbf{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 $\Gamma \vdash A : \mathbf{Type}_i$ for some universe index i . In other words, every type A always lives in some universe \mathbf{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 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 $A : \mathbf{Type}$, a constructor is a term-forming rule of the form

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

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

Clarifying note. At this stage, the expression $c(t_1, \dots, t_n)$ 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. $c(t_1, \dots, t_n)$ 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

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

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 $A \rightarrow B$, this is a *rule of the calculus*: it *licenses* forming $c(t_1, \dots, t_n)$ 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 makes sense, since by definition the empty type is empty.
- **Sum type.** For two types $A, B : \mathbf{Type}$, the sum type $A + B$ has two constructors:

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

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 $\text{inl}(a)$ for some $a : A$, or as $\text{inr}(b)$ 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.

10 Non-dependent sum types and the inl and inr constructors

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

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

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

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

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 $\text{inl}(a)$ for some $a : A$, or $\text{inr}(b)$ 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:

$$A \equiv \{\text{red}, \text{green}\}, \quad B \equiv \{0, 1\}.$$

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

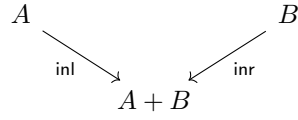
$$A + B = \{\text{inl}(\text{red}), \text{inl}(\text{green}), \text{inr}(0), \text{inr}(1)\}.$$

Intuitively, $\text{inl}(a)$ means “a value coming from A , left side of the sum,” and $\text{inr}(b)$ 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 $\text{inl}(a)$ for some $a : A$, or $\text{inr}(b)$ 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 non-dependent sum type $A + B$

Formation.

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash B : \text{Type}_i}{\Gamma \vdash A + B : \text{Type}_i} \text{+-FORM}$$

Introduction.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{inl}(a) : A + B} \text{+-INTRO}_1 \quad \frac{\Gamma \vdash b : B}{\Gamma \vdash \text{inr}(b) : A + B} \text{+-INTRO}_2$$

Elimination (case analysis).

$$\frac{\Gamma \vdash e : A + B \quad \Gamma, x:A \vdash c : C \quad \Gamma, y:B \vdash d : C}{\Gamma \vdash \text{case}(e, \lambda x. c, \lambda y. d) : C} \text{+-ELIM}$$

Computation (β -rules).

$$\frac{\Gamma \vdash a : A \quad \Gamma, x:A \vdash c : C \quad \Gamma, y:B \vdash d : C}{\Gamma \vdash \text{case}(\text{inl}(a), \lambda x. c, \lambda y. d) \equiv c[a/x] : C} \text{+-COMP}_1$$

$$\frac{\Gamma \vdash b : B \quad \Gamma, x:A \vdash c : C \quad \Gamma, y:B \vdash d : C}{\Gamma \vdash \text{case}(\text{inr}(b), \lambda x. c, \lambda y. d) \equiv d[b/y] : C} \text{+-COMP}_2$$

Explanation.

- $A + B$ is the type whose terms are either of the form $\text{inl}(a)$ with $a : A$, or $\text{inr}(b)$ with $b : B$.
- The *elimination rule* introduces the case distinction: to define a term of type C from $e : A + B$, it suffices to specify what to do in each case.
- The *computation rules* express that the case analysis reduces to the corresponding branch when the argument is a canonical injection.

11 Non-dependent product types and the pair constructor

Definition of the product type

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

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

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:

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

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:

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

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

$$A \equiv \{\text{red}, \text{green}\}, \quad B \equiv \{0, 1\}.$$

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

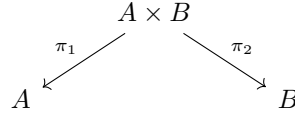
$$A \times B = \{(\text{red}, 0), (\text{red}, 1), (\text{green}, 0), (\text{green}, 1)\}.$$

This can be visualized in a grid form:

\times	0	1
red	(red, 0)	(red, 1)
green	(green, 0)	(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.

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

Introduction (pairing).

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

Elimination (projections).

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

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.

12 Functions and the arrow type ($A \rightarrow B$)

A *function* from a type A to a type B is a term of the *function type* $A \rightarrow B$. 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 $A \rightarrow B$ is a type:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A \rightarrow B : \text{Type}} \rightarrow\text{-FORM}$$

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

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

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 $f : A \rightarrow B$ and an argument $a : A$, we may *apply* f to a :

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B} \rightarrow\text{-ELIM}$$

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

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

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:

$$\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B} \rightarrow\text{-ELIM}$$

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.

$$\text{id}_A := \lambda x : A. x : A \rightarrow A, \quad \text{const}_{A,B}(a) := \lambda _ : B. a : B \rightarrow A.$$

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

Conditional function. We define the higher-order conditional operator

$$\text{if}_A := (\lambda b : \text{Bool}. (\lambda t : A. (\lambda f : A. \text{case}(b, \lambda _ . t, \lambda _ . f)))) : \text{Bool} \rightarrow (A \rightarrow (A \rightarrow A)).$$

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

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

so that

$$\text{if}_A \text{ true } t f \equiv t, \quad \text{if}_A \text{ false } t f \equiv f.$$

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

$$f \circ g := \lambda x:A. f(gx) : A \rightarrow C.$$

13 Dependent function types (Π -types)

So far we have introduced non-dependent functions $A \rightarrow B$, where both the domain A and codomain B are fixed types. In the dependent setting, however, the result type may itself depend on the input. To capture this, we introduce the *dependent function type*, written

$$\Pi(x:A).B,$$

where B may depend on the particular value $x : A$. Intuitively, an element of $\Pi(x:A).B$ is a function that, given an input $a : A$, returns a term of type $B[a/x]$.

Motivation

A non-dependent function $A \rightarrow B$ can be seen as a special case of a dependent function: if B does not depend on $x : A$, then

$$\Pi(x:A).B \equiv A \rightarrow B.$$

Thus, Π -types generalize ordinary function types.

Example. Suppose $A \equiv \text{Bool}$ and define a family of types

$$B(x) := \begin{cases} \text{Nat} & \text{if } x = \text{true}, \\ \text{String} & \text{if } x = \text{false}. \end{cases}$$

Then

$$\Pi(x:\text{Bool}).B(x)$$

is the type of functions that return a natural number when given `true` and a string when given `false`. In this case, we can think of

$$\Pi(x:\text{Bool}).B(x) \equiv B(\text{true}) \times B(\text{false}) \equiv \text{Nat} \times \text{String},$$

a “heterogeneous product” where each branch has a possibly different codomain.

Rules for the dependent function type

Formation. If A is a type and B is a type that may depend on $x : A$, then the dependent function type $\Pi(x:A).B$ is a type:

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x:A \vdash B : \text{Type}_i}{\Gamma \vdash \Pi(x:A).B : \text{Type}_i} \text{ } \Pi\text{-FORM}$$

Introduction (dependent lambda abstraction). If under the assumption $x : A$ we can derive $b : B$, then we may form a dependent function:

$$\frac{\Gamma, x:A \vdash b : B}{\Gamma \vdash \lambda(x:A).b : \Pi(x:A).B} \text{ } \Pi\text{-INTRO}$$

Here $\lambda(x:A).b$ binds all free occurrences of x in b .

Elimination (application). Given a dependent function $f : \Pi(x:A).B$ and an argument $a : A$, we can apply f to a :

$$\frac{\Gamma \vdash f : \Pi(x:A).B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]} \text{ } \Pi\text{-ELIM}$$

The result of applying f to a lies in the instance of B where every occurrence of x is replaced by a .

Computation (β -rule). When a function introduced by $\lambda(x:A).b$ is applied to an argument a , it computes by substitution:

$$\frac{\Gamma, x:A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x:A).b)(a) \equiv b[a/x] : B[a/x]} \text{ } \Pi\text{-COMP}$$

This generalizes the ordinary β -reduction rule of non-dependent functions.

Remarks

- The expression $\lambda(x:A).b$ binds all free occurrences of x in b .
- If B does not depend on x , we recover the ordinary function type $A \rightarrow B$.
- The computation rule ensures that applying a dependent function to an argument performs substitution in the codomain type.

In summary, the dependent function type $\Pi(x:A).B$ represents the type of functions that return a result of type $B[a/x]$ depending on their input $a : A$.

14 Dependent pair types (Σ -types)

A dependent pair (or Σ -type) generalizes the non-dependent product $A \times B$ to the setting where the second component may depend on the first. Given a type A and a family B of types indexed by $x : A$, we write

$$\Sigma(x:A) B$$

for the type of pairs (a, b) with $a : A$ and $b : B[a/x]$.

Rules for the dependent pair type

Formation.

$$\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x:A \vdash B : \text{Type}_i}{\Gamma \vdash \Sigma(x:A) B : \text{Type}_i} \Sigma\text{-FORM}$$

Introduction (pairing).

$$\frac{\Gamma, x:A \vdash B : \text{Type}_i \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma(x:A) B} \Sigma\text{-INTRO}$$

Elimination (dependent eliminator / induction). Let the *motive* C be a family depending on a pair $z : \Sigma(x:A) B$. We write bindings as $z.C$ to indicate z is bound in C , and $x, y.g$ to indicate x and y are bound in g . Then:

$$\frac{\Gamma, z:\Sigma(x:A) B \vdash C : \text{Type}_i \quad \Gamma, x:A, y:B \vdash g : C[(x, y)/z] \quad \Gamma \vdash p : \Sigma(x:A) B}{\Gamma \vdash \text{ind}_{\Sigma(x:A) B}(z.C, x, y.g, p) : C[p/z]} \Sigma\text{-ELIM}$$

Computation (β -rule). On a canonical pair, the eliminator computes by substituting the components:

$$\frac{\Gamma, z:\Sigma(x:A) B \vdash C : \text{Type}_i \quad \Gamma, x:A, y:B \vdash g : C[(x, y)/z] \quad \Gamma \vdash a' : A \quad \Gamma \vdash b' : B[a'/x]}{\Gamma \vdash \text{ind}_{\Sigma(x:A) B}(z.C, x, y.g, (a', b')) \equiv g[a'/x, b'/y] : C[(a', b')/z]} \Sigma\text{-COMP}$$

Binding and notation

- The expression $\Sigma(x:A) B$ *binds* x in B ; free occurrences of x in B refer to the first component of the pair.
- In the eliminator, $z.C$ binds z in C , and $x, y.g$ bind x and y in g . We treat $\text{ind}_{\Sigma(x:A) B}$ as a primitive with these binders.

Non-dependent special case

When B does not contain x free, the dependent pair collapses to the ordinary (cartesian) product; we *define*

$$A \times B \equiv \Sigma(x:A) B.$$

In this case, the introduction rule specializes to pairing $(a, b) : A \times B$, and the eliminator yields the usual projections (derivable).