Chapter 6

The Core Dependent Type Theory with Π-types

It is by the application of dependent types, which one might conceive of as a family of types, that we can extend the familiar concepts of functional and product types as found within STLC. Indeed, the Π-types and Σ-types — the former of which we shall elucidate in the current chapter, and the latter in the subsequent one — are nothing less than profound generalisations of their STLC antecedents.

Of particular note is the Π-type, which forms the very bedrock, the fundamental structure, of dependent type theory itself. This chapter, therefore, shall be dedicated to presenting a minimal organisation of dependent type theory, with the Π-type taking centre stage. This exposition will serve as the foundation for our discussions, not only in the forthcoming chapter but also in all subsequent explorations.

6.1 Π-types (or Dependent function types)

The notion of Π-types, also known as *dependent function types*, represents an extension of the more familiar function types found within STLC. While an STLC function type takes the form $A \to B$, a Π-type introduces dependency: the codomain B is not fixed, singular type, but rather a family of types, due to the free occurrence of x in B, the intuition behind which in set-theoretic notation is the union $\{B(x)\}_{x \in A}$. Moreover, functions inhabiting such a Π type are not at liberty to map an element x of type A to just any element of $\bigcup_{x \in A} \{B(x)\}$. Instead, a constraint is imposed such that x must be mapped exclusively to an element belonging to the type $B(x)$ associated with that very argument. This distinction underpins the dependent nature of these functions.[*1]

Lists

An example of a function subject to these constraints is **zeros**, which maps a natural number n to a list containing n occurrences of the numeral 0. In the context of purely

[*1] That functions belonging to a Π-type conform to the aforementioned strictures is assured solely by the typing rules which we shall deliniate in the subsequent section. The readers ought to bear in mind, that the characteristics of Π-types heretofore described above serve merely as their intuitive comprehension, rather than constituting any part of their formal definition.

functional programming paradigm, its defintion would be rendered thus:

$$\mathbf{zeros}(0) = \mathbf{nil}$$
$$\mathbf{zeros}(s(n)) = \mathbf{cons}\ 0\ \mathbf{zeros}(n)$$

In STLC, the function under consideration would typically be assigned the type $Nat \to \mathbf{List}_{Nat}$. However, this formulation is somewhat imprecise: A more faithful representation of the function's behavior necessitates capturing the explicit dependency of the output list's length on the input natural number. Specifically, for an input $n : Nat$, the function yields a list of natural numbers of length n, denoted $\mathbf{List}_{Nat}(n)$. This richer information can be elegantly encoded through the employment of dependent types, by which the type of the function transforms into $(n : Nat) \to \mathbf{List}_{Nat}(n)$, which is an instance of a Π-type.

A crucial distinguishing feature of Π-types, in contrast to the STLC function types, is the explicit binding of a variable (in this case, n) within the domain. This bound variable subsequently governs the structure of the codomain, so that the codomain of a Π-type is not static but rather varies parametrically with the input value, a fundamental departure from the fixed codomains characteristic of STLC function types.

For example, the function **zeros** would yield, for an input of 3, a list of three zeros, $[0, 0, 0]$, which is precisely of type $\mathbf{List}_{Nat}(3)$. Similarly, an input of 5 would result in $[0, 0, 0, 0, 0]$, a value of type $\mathbf{List}_{Nat}(5)$. By representing the type of such a function as a Π-type, thereby enhancing the overall precision of our type system.

Also, if the family $B(x)$ of types does not contain x as a free variational term, then the Π type merges into the functional type of STLC, since this family is a single domain consisting only of B. In this sense, the Π-type is a natural extension of the STLC functional type.

Calendar

Another a seemingly straightforward example of Π-types is the scheduling of monthly events, such as a payday. Our initial inclination might be to model this within the STLC as a function mapping months to days, perhaps of type **Month** → **Day**. This strikes us as intuitively correct, capturing the notion that for each month, there is a designated day. However, a moment's reflection reveals a crucial subtlety. As we've explored previously (and as detailed in Section Subsection 5.1), the very nature of the codomain – the set of possible days – is intrinsically linked to the specific month in question. A day in February, for instance, operates under different constraints than a day in March. The type of the "day" isn't static; it depends on the month.

This dependency is precisely what the Π-type is designed to capture. The payday, rather than being a simple function from **Month** to a fixed **Day** type, is more accurately understood as a function that, given a month m (a term of type **Month**), yields a term of type $\mathbf{Day}(m)$. This distinction is captured by the type $(m : \mathbf{Month}) \to \mathbf{Day}(m)$.[*2]

Thus, we arrive at a more sophisticated understanding: the Π-type, denoted $(x : A) \to B(x)$, serves as a generalization of the standard functional type $A \to B$.

[*2] While the standard notation in the literature is $(\Pi m : \mathbf{Month})\mathbf{Day}(m)$, the underlying concept remains the same: the output type is dependent on the input term.

Indeed, the latter is merely a special instance of the former, occurring precisely when the type B exhibits no dependence on the specific term x drawn from A (i.e., when $x \in fv(B)$). This insight underscores the power and flexibility of dependent types in formally representing the intricate dependencies inherent in various domains.

6.2 Syntax

6.2.1 Preterms

As established in Subsection 5.2.4, our initial endeavor is the rigorous definition of preterms. This collection, encompassing both type and term syntax, necessitates foundational assumptions. Analogous to our treatment within STLC, we presuppose the prior existence of a designated collection of variables ($\mathcal{V}ar$) and a distinct collection of constant symbols ($\mathcal{C}on$).

Definition 129 (Preterms)**.** A collection of *preterms* (notation Λ) under $(\mathcal{V}ar, \mathcal{C}on)$ is recursively defined by the following BNF grammar (where $x \in \mathcal{V}ar$ and $c \in \mathcal{C}on$):

$$\Lambda \quad ::= \quad x \mid c \mid \mathsf{type} \mid \mathsf{kind} \mid (x : \Lambda) \to \Lambda \mid \lambda x.\Lambda \mid \Lambda\Lambda$$

Of the syntactic constructions that appears in the definition above, x is a variable, i.e. an element of $\mathcal{V}ar$. c is a constant symbol, i.e. an element of $\mathcal{C}on$. This definition of preterm adopts the "ambitious definition" of the two approaches discussed in the previous chapter. Therefore, it is a prerequisite that all type predicates be treated as constant symbols and, as such, be contained into $\mathcal{C}on$ prior to the subsequent definitions.

The preterms type and kind are designated as *sorts*. Specifically, type is the sort for all types discussed in Subsection 5.2.2, while kind is the sort of type itself.

$(x : \Lambda) \to \Lambda$ is the syntax form of the Π-type and is a generalization of the function type $\tau \to \tau$ found in STLC[*3]. $\lambda x.\Lambda$ (λ-*abstration*) and $\Lambda\Lambda$ (*funtional application*) are proof terms for its introduction and elimination rules (see Subsection 6.4.3).

6.2.2 Free variables

For any preterm M, the set of free variables (notation: $fv(M)$) is defined by the following recursive rules.

[*3] We depart here from conventional notations in the literature. For Π, Σ, we adopt the following DTS-style notations.

$$(x : A) \to B(x) \stackrel{def}{\equiv} (\Pi x : A)B(x)$$
$$\left[\begin{array}{c} x : A \\ B(x) \end{array}\right] \stackrel{def}{\equiv} (\Sigma x : A)B(x)$$

The Π-types above are in Agda-style notation, and the Σ-types above look more like record types (cf. Type Theory with Record (TTR): Cooper (2005), Luo (2009)). The advantage of this notation will become clear in most of the semantic representations of natural language.

Definition 130 (Free variables)**.**

$$
\begin{aligned}
fv(x) &\stackrel{def}{\equiv} \{x\} \\
fv(c) &\stackrel{def}{\equiv} \varnothing \\
fv(\mathsf{type}) &\stackrel{def}{\equiv} \varnothing \\
fv(\mathsf{kind}) &\stackrel{def}{\equiv} \varnothing \\
fv((x:A) \to B) &\stackrel{def}{\equiv} fv(A) \cup (fv(B) - \{x\}) \\
fv(\lambda x.M) &\stackrel{def}{\equiv} fv(M) - \{x\} \\
fv(MN) &\stackrel{def}{\equiv} fv(M) \cup fv(N)
\end{aligned}
$$

A variable x comprises a singleton set of free variables, $\{x\}$. Constant symbols and sorts do not contain any free variables.

A set of free variables of a perterm is basically defined as a union of sets of free variables of its parts. An exception is a *binder* such as $(x : A) \to$ and λx, which binds a variable that appears free in its scope, so that the variable in question is not a free variable in a larger construction.

A term M is a *closed term* iff $fv(M) = \varnothing$. Otherwise it is an *open term*.

Implication in propositional logic and first-order logic is defined as a special case of Π-type as follows.

Definition 131 (Implication)**.** $A \to B \stackrel{def}{\equiv} (x : A) \to B \qquad$ *where* $x \notin fv(B)$.

6.2.3 Substitution

The notation for substitution $M[L/x]$, where L, M are preterms and x is a variable (which may or may not appear in M), reads as the term M where a free occurrences of x is substituted by L. The substitution operation is recursively defined by the following set of rules.

Definition 132 (Substitution)**.**

$$
\begin{array}{rcl}
x[L/x] & \stackrel{def}{\equiv} & L \\
y[L/x] & \stackrel{def}{\equiv} & y \\
c[L/x] & \stackrel{def}{\equiv} & c \\
\mathsf{type}[L/x] & \stackrel{def}{\equiv} & \mathsf{type} \\
\mathsf{kind}[L/x] & \stackrel{def}{\equiv} & \mathsf{kind} \\
((x:A) \to B)[L/x] & \stackrel{def}{\equiv} & (x : A[L/x]) \to B \\
((y:A) \to B)[L/x] & \stackrel{def}{\equiv} & (y : A[L/x]) \to (B[L/x]) \quad \textit{where } x \notin fv(B) \textit{ or } y \notin fv(L). \\
(\lambda x.M)[L/x] & \stackrel{def}{\equiv} & \lambda x.M \\
(\lambda y.M)[L/x] & \stackrel{def}{\equiv} & \lambda y.M[L/x] \quad \textit{where } x \notin fv(M) \textit{ or } y \notin fv(L). \\
(MN)[L/x] & \stackrel{def}{\equiv} & (M[L/x])(N[L/x])
\end{array}
$$

$x[L/x]$ is L since substituting L for x in x yields L, while $y[L/x]$ is just y since there is no x in y. In the same way, $c[L/x], \mathsf{type}, \mathsf{kind}$ do not change by any substitution since there is no x therein.

When a variable bound by Π or Σ coincides with the variable of substitution – e.g., in $((x : A) \to B)[L/x]$ or in $(\lambda x.M)[L/x]$, – no substitution occurs.

Otherwise, substitution proceeds recursively, as exemplified by $((y : A) \to B)[L/x]$ and $(\lambda y.M)[L/x]$. For such a substitution to be permissible, a crucial side condition must be satisfied: specifically, $x \notin fv(B)$ or $y \notin fv(L)$ for the Π case, and $x \notin fv(M)$ or $y \notin fv(L)$ for the λ case. Recall the exposition following Definition 47 for a justification for these conditions. Should these conditions not be met, the variable on the binder side must be converted using the α-conversion rules defined as follows.

Definition 133 (α-conversion rules)**.**

$$
\begin{array}{c}
(x : A) \to B \equiv (y : A) \to B[y/x] \quad \textit{where } y \notin fv(B). \\
\lambda x.M \equiv \lambda y.(M[y/x]) \quad \textit{where } y \notin fv(M).
\end{array}
$$

Theorem 134. For any preterms A and M, $A \equiv A[M/x]$, if $x \notin fv(A)$.

Proof. By induction on the depth of A. □

Exercise 135. Prove Theorem 134.

6.3 Context and Judgment

A *type assignment* is the form $M : A$ where M and A are preterms, stating that "a term M is of type A". Signatures are the structures which specify the type of each constant symbol.

Definition 136 (Signature)**.** A collection of *signatures* (notation σ) under $(\mathcal{V}ar, \mathcal{C}on)$ is recursively defined by the following BNF grammar:

$$\sigma ::= (\ ) \mid \sigma,\ c : A$$

where $(\ )$ is an empty signature, $c \in \mathcal{C}on$, and A is a preterm such that $\vdash_\sigma A : \mathsf{type}$ or $\vdash_\sigma A : \mathsf{kind}$.

We ofter abbreviate the signature $(), c : A$ as $c : A$.

The definition of signature includes the relation (judgment) represented by $\vdash_\sigma$, which is defined by Definition 139, and judgment is defined by the typing rules in the next section. Thus, signature, judgment, typing rules, (and β-conversions) in DTT are mutually recursively defined concepts.

Contexts are the structures which specify the type of each variable.

Definition 137 (Context)**.** A collection of *contexts* (notation Γ) under a signature σ is recursively defined by the following BNF grammar:

$$\Gamma ::= (\ ) \mid \Gamma,\ x : A$$

where $(\ )$ is an empty context, $x \in \mathcal{V}ar$, and A is a preterm such that $\Gamma \vdash_\sigma A : \mathsf{type}$ or $\Gamma \vdash_\sigma A : \mathsf{kind}$.

We ofter abbreviate the context $(), M : A$ as $M : A$.

The condition on A is referred to as the "*condition on variables*" (henceforth *c.o.v.*).

Exercise 138. What would happen if the system does not require c.o.v.?

Definition 139 (Judgment)**.** The following form:

$$\Gamma \vdash_\sigma M : A$$

is called *judgment* stating that there exists a *proof diagram* from the context Γ to the type assignment $M : A$ under the signature σ. In particular, the judgment of the form:

$$\Gamma \vdash_\sigma A\ \mathit{true}$$

states that there exists a term M that satisfies $\Gamma \vdash_\sigma M : A$.

We often omit σ from a judgment and just write it as $\Gamma \vdash M : A$ when no confusion arises.

As previously established, the class of preterms encompasses expressions of a hybrid nature, such as $(x : A) \to B$, wherein A or B is not a type under the given context. Such heterodox constructions necessitate exclusion from the domain of meaningful expressions.

Consequently, we formally delineate the notion of well-formed types as a proper subset of well-formed terms. This restricted collection comprises precisely those preterms A for which the judgment $\Gamma \vdash A : \mathsf{type}|\mathsf{kind}$ is derivable through the application of the typing rules. These rules, operating under a given context Γ, will be formally

introduced in the ensuing sections.

6.4 Typing Rules

6.4.1 Axioms

An axiom is an inference rule with no premise. There are two axiom schemata in DTT.

Definition 140 (Constant symbols)**.**

$$\frac{}{c : A}\,{}^{(CON)} \quad \textit{where } (c : A) \in \sigma.$$

Let $(c : A) \in \sigma$ be (not in set theory notation) that the type assignment $c : A$ is contained in the signature σ.

The second axiom is the following rule which states that type is a preterm of sort kind.

Definition 141 (Type of types)**.**

$$\frac{}{\mathsf{type} : \mathsf{kind}}\,{}^{(\mathsf{type}F)}$$

To form a signature σ, we first need a type assignment that can be made without assumptions, and the $(\mathsf{type}F)$ rule provides that. Let us look at the following example.

Example 142. Here is an example of introducing a constant symbol using signature: In STLC, if A is a type, we can assume $x : A$. In DTT, however, we have to establish $A : \mathsf{type}$ or $A : \mathsf{kind}$ in the signature prior to assume $x : A$ (i.e. add $x : A$ to the context). According to Definition 137, we must obtain

$$(), A : \mathsf{type} \text{ is a signature}$$

from the following.

$$() \text{ is a signature}$$

For this step, we first need $A \in \mathcal{C}on$, which we now assume to be the case. Next, by c.o.v., we need $\vdash \mathsf{type} : \mathsf{kind}$. This must be derived from no assumptions, and this is where the $(\mathsf{type}F)$ rule is used.

Exercise 143. Confirm that there is no way to add $A : \mathsf{kind}$ to the signature, for any $A \in \mathcal{C}on$.

6.4.2 Structural Rules

The following rule is called *weakening* rule, which dischards one of the two premises.

Theorem 144 (Weakening)**.**

$$\frac{M : A \quad N : B}{M : A}\,(WK)$$

The following $(CONV)$ rules state that if a term M has a type A and A is β-equivalent to another type B, then M also has a type B. The definition of the β-equivalence $=_\beta$ will be given by Definition **??**.

Definition 145 (Conversion Rule)**.**

$$\frac{M : A}{M : B}\,(CONV) \quad \textit{where } A =_\beta B.$$

6.4.3 Π-types

Let $\diamondsuit$ denote an arbitrary type constructor within the framework of DTT. The 'meaning' of any complex type formed by $\diamondsuit$, or equivalently, its behaviour within proof diagrams, is defined by the tripartite set of rules: *formation rules*, *introduction rules* and *elimination rules* (notation: $(\diamondsuit F)$, $(\diamondsuit I)$ and $(\diamondsuit E)$). For instance, in the case of Π-types, their meaning and behaviour are specified by the (ΠF), (ΠI) and (ΠE) rules.

Formation rules states under which condition a complex type in question is well-formed, and when it is well-formed, it is of which sort, namely, **type** or **kind**.

Definition 146 (Π-Formation Rule)**.**

$$\frac{\begin{array}{c}\overline{x : A}^{\,i}\\ \vdots\\ A : \mathrm{s}_1 \quad B : \mathrm{s}_2\end{array}}{(x : A) \to B : \mathrm{s}_2}\,(\Pi F),i \quad \textit{where } (\mathrm{s}_1, \mathrm{s}_2) \in \left\{ \begin{array}{l} (\mathsf{type}, \mathsf{type}),\\ (\mathsf{type}, \mathsf{kind}),\\ (\mathsf{kind}, \mathsf{kind}) \end{array} \right\}.$$

The collection of preterms includes those of the form $(x : A) \to B$. However, their well-formedness as types is contingent upon a given signature and context. The (ΠF), the formation rule for the Π-types, delineates the conditions under which a preterm of the form $(x : A) \to B$ attains the status of a well-formed type. That is, for $(x : A) \to B$ to constitute a sort, it is a prerequisite that both A and B themselves be sorts. This constraint mirrors a similar restriction found in the syntactic definition of types within SLTC, where the construct $\tau \to \tau$ is a well-formed type only if two occurrences of τs are types.

This strategy, namely, eliminating the distinction between type and term in syntax and defining a type as "a term M such that $\Gamma \vdash M : \mathsf{type}$ is shown by inference to be a type under the environment Γ" differs from global types in the STLC concept.

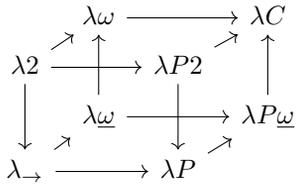
In the definition of (ΠF), the following part shows what sorts s_1 and s_2 can range over.

$$where\ (\mathrm{s}_1, \mathrm{s}_2) \in \left\{ \begin{array}{l} (\mathrm{type}, \mathrm{type}), \\ (\mathrm{type}, \mathrm{kind}), \\ (\mathrm{kind}, \mathrm{kind}) \end{array} \right\}.$$

For this, a different constraint than the above definition could be given, which yields different DTT systems depending on how this constraint is given. Each DTT system is named as follows.

$$\begin{array}{lcl} \lambda_{\to} & : & \{(\mathrm{type}, \mathrm{type})\} \\ \lambda P & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{type}, \mathrm{kind})\} \\ \lambda 2 & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{kind}, \mathrm{type})\} \\ \lambda \underline{\omega} & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{kind}, \mathrm{kind})\} \\ \lambda P2 & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{type}, \mathrm{kind}), (\mathrm{kind}, \mathrm{type})\} \\ \lambda P\underline{\omega} & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{type}, \mathrm{kind}), (\mathrm{kind}, \mathrm{kind})\} \\ \lambda \omega & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{kind}, \mathrm{type}), (\mathrm{kind}, \mathrm{kind})\} \\ \lambda C & : & \{(\mathrm{type}, \mathrm{type}), (\mathrm{type}, \mathrm{kind}), (\mathrm{kind}, \mathrm{type}), (\mathrm{kind}, \mathrm{kind})\} \end{array}$$

These systems form the following hierarchy by constraint inclusion relations, which we call λ-cube (or the Barendregt Cube, Barendregt (1992)) or more generally Pure Type Systems (PTS) (Berardi (1990); Barendregt (1991)).



Next, consider the introduction rule.

Definition 147 (Π-Introduction Rule)**.**

$$\frac{\begin{array}{c} \overline{x : A}^{\,i} \\ \vdots \\ A : \mathrm{s} \quad M : B \end{array}}{\lambda x.M : (x : A) \to B}\ (\Pi I), i \qquad where\ \mathrm{s} \in \{\mathrm{type}, \mathrm{kind}\}.$$

The Π-introduction rules states the *verification condition* of Π-types, namely, under what condtion the type $(x : A) \to B$ inhabits a term, or equivalently, under what condition the proposition $(x : A) \to B$ is true, under a given context. In this case, $(x : A) \to B$ inhabits a term $\lambda x.M$ iff A is a sort and $M : B$ is derived from the assumption $x : A$.

Another role of the Π-introduction rule is that it gives a well-formed condition for the complex term $\lambda x.M$, in the sense that the verification condition of $(x : A) \to B$ is exactly the condition for the term $\lambda x.M$ to be well-formed.

Finally, consider the elimination rule.

Definition 148 (Π-Elimination Rule)**.**

$$\frac{M : (x : A) \to B \quad N : A}{MN : B[N/x]}\ (\Pi E)$$

The Π-elimination rule states the *pragmatic condition* of Π-types, namely, how a term of type $(x : A) \to B$ can be *used* in proof diagrams, or equivalently, what conclusion derives from the truth of a proposition $(x : A) \to B$. In this case, a term of type $(x : A) \to B$ can be used to derive a term of type $B[N/x]$, if N, a proof of type A, is given.

In other words, the introduction rule (ΠI) and the elimination rule (ΠE) respectively determine from what a type of the form $(x : A) \to B$ is derived, and what derives from the type of the form $(x : A) \to B$, which in turn determine the 'meaning' or the 'behaviour' of the type $(x : A) \to B$.

Theorem 149 (Derived rules for implication)**.** The following rules are derivable where $x \notin fv(B)$.

$$\frac{\begin{array}{c}\overline{x : A}^{\,i}\\ \vdots\\ A : \mathsf{type} \quad M : B\end{array}}{\lambda x.M : A \to B}\ (\to I),i \qquad \frac{M : A \to B \quad N : A}{MN : B}\ (\to E)$$

Proof. $(\to I)$ is the same as (ΠI). $(\to E)$ is a direct consequence of (ΠE) and Theorem 134. □

Example 150. In the previous chapter, we noted that for dependent types, these inference rules interact with the type definition. Take the calender example again. Let **Month** and **Day** be defined as elements of $\mathcal{C}on$. Then, the signature is composed as follows.

$$\mathbf{Month} : \mathsf{type},\ \mathbf{Day} : \mathbf{Month} \to \mathsf{type}$$

Let's confirm that this configuration satisfies the condition on variable (c.o.v). First, since $\mathbf{Month} \in \mathcal{C}on$ and $\vdash \mathsf{type} : \mathsf{kind}$, $\mathbf{Month} : \mathsf{type}$ is a well-formed signature. Next, under this signature, we can show that $\mathbf{Month} : \mathsf{type},\ \mathbf{Day} : \mathbf{Month} \to \mathsf{type}$ is a well-formed signature, since the following holds.

$$\frac{\dfrac{}{\mathbf{Month} : \mathsf{type}}\ (CON) \qquad \dfrac{}{\mathsf{type} : \mathsf{kind}}\ (\mathsf{type}F)}{\mathbf{Month} \to \mathsf{type} : \mathsf{kind}}\ (\Pi I)$$

Furthermore, under this second signature, it can be shown that $\vdash (x : \mathbf{Month}) \to \mathbf{Day}(x) : \mathsf{type}$.

$$\frac{\dfrac{}{\mathbf{Month} : \mathsf{type}} \qquad \dfrac{\dfrac{}{\mathbf{Day} : \mathbf{Month} \to \mathsf{type}}\ (CON) \qquad \dfrac{}{x : \mathbf{Month}}\ 1}{\mathbf{Day}(x) : \mathsf{type}}\ (\Pi E)}{(x : \mathbf{Month}) \to \mathbf{Day}(x) : \mathsf{type}}\ (\Pi E),1$$

6.5 Reduction

The *reduction* rules are, from the logical perspective, the rules for normalization of proofs, and from the type-theoretical perspective, the rules for computation or execusion of programs. Specifically, the *redex rule* as presented in Definition 151 deliniates a *detour* of proof diagrams: the immediate application of the (ΠE) rule following the (ΠI). Such detours are inherently normalizable. Furthermore, the *structure rules* articulated in Definition 151 establish that the one-step β-reduction is permissible within any subformula of the preterms.

Definition 151 (One-step β-reduction)**.**

Redex rule

$$(\lambda x.M)N \to_\beta M[N/x]$$

Structure rules

$$\frac{A \to_\beta A'}{(x : A) \to B \to_\beta (x : A') \to B} \qquad \frac{B \to_\beta B'}{(x : A) \to B \to_\beta (x : A) \to B'}$$

$$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'} \qquad \frac{M \to_\beta M'}{MN \to_\beta M'N} \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}$$

Remark 152. Multi-step β-reduction, denoted as $\twoheadrightarrow_\beta$, is defined analogously to Definition 57. It share the fundamental properties with its counterpaert in STLC, and exhibits both reflexivity and transitivity. Furthermore, β-equivalence, denoted as $=_\beta$, when defined following Definition 61, constitutes an equivalence relation.

Lemma 153 (Substitution Lemma I)**.** For any preterms M, M', N and a variable x, the following properties hold:

$$M \to_\beta M' \implies M[N/x] \to_\beta M'[N/x]$$

Exercise 154. Prove Lemma 153 by induction on the depth of the proof of $M \to_\beta M'$.

Lemma 155 (Substitution Lemma II)**.** For any preterms M, N, N' and a variable x, the following properties hold:

$$N \to_\beta N' \implies M[N/x] \to_\beta M[N'/x]$$

Exercise 156. Prove Lemma 155 by induction on the depth of the proof of $N \to_\beta N'$.

Lemma 157 (Substitution Lemma III)**.** For any preterms M, M', N, N' and a variable x, the following properties hold:

1. $M =_\beta M' \implies M[N/x] =_\beta M'[N/x]$
2. $N =_\beta N' \implies M[N/x] =_\beta M[N'/x]$

Exercise 158. Prove Lemma 157 by induction on the depth of the proof of β-equivalence.

6.6 Subject Reduction Theorem

Lemma 159 (Substitution Lemma IV)**.** For any preterm L, M, A, C and a variable z, the following property hold.

$$\Gamma \vdash L : C,\ \Gamma, z : C \vdash M : A \implies \Gamma \vdash M[L/z] : A[L/z]$$

Proof. Assume there are two proof diagrams $\mathcal{D}_L$ and $\mathcal{D}_M$ that satisfy the following.

$$\begin{array}{cc} \Gamma & \Gamma, z : C \\ \mathcal{D}_L & \mathcal{D}_M \\ L : C & M : A \end{array}$$

Let $\mathrm{P}(d)$ be the proposition that the Substitution Lemma I holds for the $\mathcal{D}_M$ whose depth is equal or less than d.
Case of $\underline{d = 1}$: $M : A$ is an axiom and $z \notin M$ and $z \notin A$. Thus, by Theorem 134, $\mathrm{P}(1)$ holds.

Case of $\underline{d \geq 1}$: Let us show that $\mathrm{P}(d)$ holds, sssuming the induction hypothesis (IH) that $\mathrm{P}(1), \ldots, \mathrm{P}(d-1)$ hold, by deviding the case of the last rule used in $\mathcal{D}_M$.
$\boxplus$ (ΠI): $\mathcal{D}_M$ must be proved as follows (assuming that $x \not\equiv z$, and $z \notin fv(B)$ or $x \notin fv(L)$).

$$\mathcal{D}_M \equiv \dfrac{\begin{array}{cc} z : C & \overline{x : A, z : C}^{\,i} \\ \vdots & \vdots \\ A : \mathsf{type} & M : B \end{array}}{\lambda x.M : (x : A) \to B}\,(\Pi I), i$$

From (IH), $A[L/z] : \mathsf{type}[L/z]$ and $M[L/z] : B[L/z]$ thus we obtain the following proof diagram.

$$\dfrac{\begin{array}{cc} & \overline{x : A[L/z]}^{\,i} \\ & \vdots \\ A[L/z] : \mathsf{type} & M[L/z] : B[L/z] \end{array}}{\lambda x.M[L/z] : ((x : A) \to B)[L/z] \equiv (x : A[L/z]) \to B[L/z]}\,(\Pi I), i$$

⊞ $\underline{(\Pi E)}$: $\mathcal{D}_M$ must be proved as follows (assuming that $x \not\equiv z$, and $z \notin fv(B)$ or $x \notin fv(L)$).

$$\mathcal{D}_M \equiv \dfrac{\begin{array}{cc} z:C & z:C \\ \vdots & \vdots \\ M:(x:A)\to B & N:A \end{array}}{MN:B[N/x]}\,(\Pi E)$$

From (IH), $M[L/z] : ((x : A) \to B)[L/z]$ and $N[L/z] : A[L/z]$ thus we obtain the following proof diagram.

$$\dfrac{M[L/z]:((x:A)\to B)[L/z] \equiv (x:A[L/z])\to B[L/z] \quad N[L/z]:A[L/z]}{(MN)[L/z] \equiv M[L/z]N[L/z] : B[L/z][N[L/z]/x]}\,(\Pi E)$$

Since $x \notin fv(L)$, $B[L/z][N[L/z]/x] \equiv B[N/x][L/z]$.
⊞ $\underline{(CONV)}$: $\mathcal{D}_M$ must be proved as follows.

$$\dfrac{\begin{array}{c} z:C \\ \mathcal{D}_B \\ M:B \end{array}}{M:A}\,(CONV) \quad \textit{where } A =_\beta B.$$

By applying (IH) to $\mathcal{D}_B$, we obtain the proof of $M[L/z] : B[L/z]$. From $A =_\beta B$ and Lemma 157 (1.), $A[L/z] =_\beta B[L/z]$. Therefore, we obtain the following proof.

$$\dfrac{\begin{array}{c} z:C \\ IH(\mathcal{D}_B) \\ M[L/z]:B[L/z] \end{array}}{M[L/z]:A[L/z]}\,(CONV) \quad \textit{where } A[L/z] =_\beta B[L/z].$$

The cases of (ΠF) and (WK) are left as an exercise. □

Exercise 160. Complete the proof of Lemma 153.

Subject reduction, also known as *type preservation* (recall Theorem 110) holds for DTT.

Theorem 161 (Subject Reduction for DTT)**.** For any context Γ and preterms M, M', A, if $\Gamma \vdash M : A,\ M \to_\beta M'$, then $\Gamma \vdash M' : A$.

The proof largely mirrors that of Theorem 110, requiring extention only to accommodate the novel syntactic forms and the typing ruules introduced for DTT.

Proof. By induction on the depth of proof diagram for $M \to_\beta M'$.

Case of $\underline{d = 0}$: M is a β-redex. The cases are divided below according to the form of $M \to_\beta M'$.
⊞ $\underline{(\lambda x.M)N \to_\beta M[N/x]}$: Suppose that $(\lambda x.M)N$ has a type $B[N/x]$ for some B, which is proved as follows.

$$\dfrac{\dfrac{\mathcal{D}_A \quad \overline{x : A}^{\,i} \quad \mathcal{D}_M}{A : \mathrm{type} \quad M : B}\,(\Pi I),i \quad \mathcal{D}_N}{\dfrac{\lambda x.M : (x : A) \to B \qquad N : A}{(\lambda x.M)N : B[N/x]}\,(\Pi E)}$$

By applying Lemma 159 to $\mathcal{D}_M$ and $\mathcal{D}_N$, we obtain the following proof.

$$\begin{array}{c} \mathcal{D}_N \\ N : A \\ \mathcal{D}_{M[N/x]} \\ M[N/x] : B[N/x] \end{array}$$

Case of $\underline{d > 0}$: Suppose that Theorem 161 holds when $M \to_\beta M'$ is proven in less than d-steps (IH). Let us prove that Theorem 161 holds when $M \to_\beta M'$ is proven in d-steps.
⊞ $\underline{(x : A) \to B \to_\beta (x : A') \to B}$: Suppose that $(x : A) \to B$ has a type s_2, which is proved as follows.

$$\dfrac{\mathcal{D}_A \quad \overline{x : A}^{\,i} \quad \mathcal{D}_B}{A : \mathrm{s}_1 \quad B : \mathrm{s}_2}\,(\Pi F),i$$
$$(x : A) \to B : \mathrm{s}_2$$

and $(x : A) \to B \to_\beta (x : A') \to B$ is proved as follows.

$$\dfrac{\begin{array}{c} \mathcal{D}_\beta \\ A \to_\beta A' \end{array}}{(x : A) \to B \to_\beta (x : A') \to B}$$

Thus, we obtain the following proof.

$$\dfrac{\dfrac{\mathcal{D}_A \quad \mathcal{D}_\beta}{A : \mathrm{s}_1 \quad A \to_\beta A'}\,(IH) \qquad \dfrac{\overline{x : A'}^{\,i}}{x : A}\,(CONV) \quad \mathcal{D}_B}{A' : \mathrm{s}_1 \qquad B : \mathrm{s}_2}\,(\Pi F)$$
$$(x : A') \to B : \mathrm{s}_2$$

⊞ $\underline{MN \to_\beta MN'}$: Suppose that MN has a type $B[N/x]$, which is proved as follows.

$$\dfrac{\mathcal{D}_M \qquad \mathcal{D}_N}{M : (x : A) \to B \quad N : B}\,(\Pi E)$$
$$MN' : B[N/x]$$

and $MN \twoheadrightarrow_\beta MN'$ is proved as follows.

$$\dfrac{\begin{array}{c} \mathcal{D}_\beta \\ N \to_\beta N' \end{array}}{MN \to_\beta MN'}$$

By Substitution Lemma II, 2, $B[N/x] =_\beta B[N'/x]$ holds. Thus, we obtain the following proof.

$$\dfrac{\dfrac{\mathcal{D}_M}{M : (x : A) \to B} \quad \dfrac{\dfrac{\mathcal{D}_N}{N : B} \quad \dfrac{\mathcal{D}_\beta}{N \to_\beta N'}}{N' : B}\,(IH)}{\dfrac{MN' : B[N'/x]}{MN' : B[N/x]}\,(CONV)}\,(\Pi E)$$

The cases of $(x : A) \to B \to_\beta (x : A) \to B'$, $\lambda x.M \to_\beta \lambda x.M'$, and $MN \to_\beta M'N$ are left to the readers as an exercise. □

Exercise 162. Complete the proof of Theorem 161.

Remark 163. The reverse of this, i.e.

$$\Gamma \vdash M' : A,\ M \twoheadrightarrow_\beta M' \Longrightarrow \Gamma \vdash M : A$$

is not necessarily true. For example, $(\lambda y.\lambda x.x)(\lambda x.xx) \twoheadrightarrow_\beta \lambda x.x$ and (assuming A : type) $\vdash \lambda x.x : A \to A$ but $(\lambda y.\lambda x.x)(\lambda x.xx)$ has no type.

Remark 164. Therefore, $\Gamma \vdash M' : A$ cannot be deduced from $\Gamma \vdash M : A$ and $M =_\beta M'$.

6.7 Discussion

6.7.1 Formation rules as introduction rules

The Π-type formation rule specifies the necessary condition for a preterm of the form $(x : A) \to B$ to be of type, thereby licensing the judgment $(x : A) \to B$: type. From an alternative vintage, this very rule can be construed as an introduction rules for type.

A similar duality pervades the type rule. While serving as a formation rule for the sort type itself, it simultaneously functions as an introduction rule for encmpassing sort kind.

This perspective suggests a profound insight: the meaning of a type, even in the rich context of dependent types, is fundamentally determined by the pair of its introduction and elimination rules. Consequently, the formation rules for type and kind are, in essence, their respective introduction rules, thereby fixing their semantic content.

History and Further Reading

The concept of *dependent types* traces its genesis to Per Martin-Löf's seminal work on Martin-Löf Type Theory (MLTT) (Martin-Löf (1975, 1984)). Conceived as a foundational system for constructive mathematics, MLTT's profound insight subsequently informed the development of Calculus of Constructions (CoC) (Coquand and Huet (1988)), a system designed to undergird both functional programming paradigms and the computational formalization of mathematical proofs.

More recently, salient fragments of both MLTT and CoC have been seamlessly integrated broader theoretical frameworks. Foremost among these is the λ-cube, also known as the Barendregt Cube Barendregt (1992), and, more generally, Pure

Type Systems (PTS) (Berardi (1990); Barendregt (1991)). These unifying accommodate various other pivotal type theories, including, notably, Girard's F (Girard et al. (1989)).

The Calculus of Inductive Constructions (CoIC), which extends CoC with the mechanism of inductive types, has emerged as the foundational language for prominent proof assistants. Exemplars include *Coq* (Bertot and Castéran (2004)) and *Agda* (Nordström et al. (1990), Bove and Dybjer (2008)), systems that leverage the expressive power of dependent and inductive types to facilitate the construction and verification of complex mathematical proofs and formally verified software.

The nomenclature "Dependent Type Theory" has, in recent discourse, come to encompass two distinct yet interconnected formal systems:

1. λP: this system, residing within Barendregt's Cube, extends the foundational STLC by incorpolating Π-types.
2. Martin-Löf/Constructive/Intuitionistic/Modern Type Theory: these systems, building upon λP, introduce a richer array of constructs, including Σ-types, (dependent) record types, equational type, natural numbers, and various forms of inductive types.