LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

A Unifying Theory of Dependent Types I

by

Zhaohui Luo

A Unifying Theory of Dependent Types I

LFCS Report Series **ECS-LFCS-91-154**

LFCS **May 1991**

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

A Unifying Theory of Dependent Types I

Zhaohui Luo

April 30, 1991

1 Introduction

In these notes,[1] we present and study a theory of dependent types which unifies coherently Martin-Löf's type theory with universes [ML75, ML84] and Coquand-Huet's calculus of constructions [CH88, Coq89]. The theory described here is a direct further development of the Extended Calculus of Constructions [Luo89, Luo90]. We (partly) follow Martin-Löf's intuitionistic philosophy in that the theory is supposed to be *open* in the sense that new types and type constructors can be added, although we consider an impredicative 'universe' of logical propositions which provides a higher-order logic embedded in the theory. However, an important philosophical point is worth making clear; that is, the incorporation of both an impredicative universe (higher-order logic) and predicative type universes enables a *conceptual distinction* in the theory between the notion of logical formulas (propositions) and that of sets (types). This should be considered in contrast with Martin-Löf's type theory, where types and propositions are identified, and with the pure calculus of constructions, where data types may be provided by impredicative coding. We also consider well-ordering types (W-types) to provide various inductive data types, based on an observation that various *filling-up equality rules* (or η-rules) are sufficient and necessary for W-types to represent inductive data types. The inclusion of the filling-up rules requires a further investigation of the meta-theory (Church-Rosser, normalization, *etc.*), which is to be discussed in a supplementary note. An alternative formulation of the type system by means of Martin-Löf's logical framework is given in Appendix A.

2 Judgement Forms and Some Conventions

Terms will be introduced when we proceed to present the inference rules for various type constructors. Given the notion of terms, we take the notions of context, free/bound

[1]These are revised notes from a part of the lecture notes for the Theory PG course on Type Theory in the spring term in LFCS, Edinburgh. I'd like to thank Rod Burstall and Healfdene Goguen for many interesting discussions and Thierry Coquand, Thorsten Altenkirch and Claudio Hermida, among others, for their helpful comments on the notes.

variables, *etc.* as defined in the usual way. We shall write $FV(...)$ for the set of free variables that occur in .... Substitution of a term N for the free occurrences of variable x in any expression M (a term or a context), $[N/x]M$, is defined as usual with possible changes of bound variables to avoid variable capturing. α-convertible terms are identified. We use $\equiv$ for syntactic indentity.

We shall consider five judgement forms:[2]

- Γ **valid**, which means that Γ is a valid (well-formed) context;
- $\Gamma \vdash A$ **type**, which means that A is a type under the hypotheses Γ;
- $\Gamma \vdash A = B$, which means that types A and B are equal under the hypotheses Γ;
- $\Gamma \vdash a : A$, which means that a is of type A under the hypotheses Γ; and
- $\Gamma \vdash a = b : A$, which means that a and b are equal objects of type A under the hypotheses Γ.

Derivation and derivability of judgements are defined as usual (but see the convention in the following section).

2.1 Conventions in the presentation

In order to simplify the presentation of the inference rules for judgements, we make the following conventions to the formal system we consider below, so that some of the premises in the inference rules are not explicitly given.

1. To derive $\Gamma \vdash A$ **type**, Γ **valid** must have been derived.
2. To derive $\Gamma \vdash A = B$, $\Gamma \vdash A$ **type** and $\Gamma \vdash B$ **type** must have been derived.
3. To derive $\Gamma \vdash a : A$, $\Gamma \vdash A$ **type** must have been derived.
4. To derive $\Gamma \vdash a = b : A$, $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$ must have been derived.

In other words, a sequence of judgements $J_1, ..., J_n$ is a derivation (of J_n) if and only if

1. it satisfies the above conditions (*i.e.*, for example, if $J_i \equiv \Gamma \vdash A$ **type**, then $J_k \equiv \Gamma$ **valid** for some $k < i$), and
2. for $i = 1, ..., n$, J_i is the conclusion of some instance of an inference rule whose premises are among $\{ J_j \mid j < i \}$.

Some other conventions (*e.g.*, abbreviations of terms) are also adopted, which we shall explain in section 4, where they are used for the first time.

[2] Another form of judgements for subtyping will be considered later.

3 General Rules

3.1 Validity of contexts

$$\frac{}{\langle\rangle \textbf{ valid}} \qquad \frac{\Gamma \vdash A \textbf{ type} \quad x \notin FV(\Gamma)}{\Gamma, x{:}A \textbf{ valid}}$$

3.2 Assumptions

$$\frac{}{\Gamma, x{:}A, \Gamma' \vdash x : A}$$

Note that, according to our conventions in section 2.1, to use this rule to derive $\Gamma, x{:}A, \Gamma' \vdash x : A$, we must have derived $\Gamma, x{:}A, \Gamma'$ **valid** and $\Gamma, x{:}A, \Gamma' \vdash A$ **type**. We do not comment on such conventions any more below.

3.3 General equality rules

3.3.1 type equality

$$\frac{}{\Gamma \vdash A = A} \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A} \qquad \frac{\Gamma \vdash A = B \quad \Gamma \vdash B = C}{\Gamma \vdash A = C}$$

3.3.2 object equality

$$\frac{}{\Gamma \vdash a = a : A} \qquad \frac{\Gamma \vdash a = b : A}{\Gamma \vdash b = a : A} \qquad \frac{\Gamma \vdash a = b : A \quad \Gamma \vdash b = c : A}{\Gamma \vdash a = c : A}$$

3.4 Equality typing

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A = B}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a = a' : A \quad \Gamma \vdash A = B}{\Gamma \vdash a = a' : B}$$

4 An Impredicative Universe: Higher-order Logic

By the proposition-as-types principle, we can introduce a type *Prop* whose objects are the logical formulas (called *propositions*), each of which is the name of the type of its proofs $\mathbf{Prf}(P)$. The impredicativity of the formation of propositions gives us a higher-order logic embedded in the theory.

The following are the formation and introduction rules for the impredicative universe.

$$\frac{}{\Gamma \vdash \mathit{Prop} \textbf{ type}} \qquad \frac{\Gamma \vdash A \textbf{ type} \quad \Gamma, x{:}A \vdash P : \mathit{Prop}}{\Gamma \vdash \forall x{:}A.P : \mathit{Prop}} \qquad \frac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash P = P' : \mathit{Prop}}{\Gamma \vdash \forall x{:}A.P = \forall x{:}A'.P' : \mathit{Prop}}$$

formation	$\dfrac{\Gamma \vdash P : \mathit{Prop}}{\Gamma \vdash \mathbf{Prf}(P)\ \mathbf{type}}$	$\dfrac{\Gamma \vdash P = P' : \mathit{Prop}}{\Gamma \vdash \mathbf{Prf}(P) = \mathbf{Prf}(P')}$
introduction	$\dfrac{\Gamma, x{:}A \vdash p : \mathbf{Prf}(P)}{\Gamma \vdash \Lambda[\forall x{:}A.P]x.p : \mathbf{Prf}(\forall x{:}A.P)}$	$\dfrac{\Gamma, x{:}A \vdash p = p' : \mathbf{Prf}(P)}{\Gamma \vdash \Lambda x.p = \Lambda x.p' : \mathbf{Prf}(\forall x{:}A.P)}$
elimination	$\dfrac{\Gamma \vdash p : \mathbf{Prf}(\forall x{:}A.P)\ \ \Gamma \vdash a : A}{\Gamma \vdash \mathbf{App}[\forall x{:}A.P](p,a) : \mathbf{Prf}([a/x]P)}$	$\dfrac{\Gamma \vdash p = p' : \mathbf{Prf}(\forall x{:}A.P)\ \ \Gamma \vdash a = a' : A}{\Gamma \vdash \mathbf{App}(p,a) = \mathbf{App}(p',a') : \mathbf{Prf}([a/x]P)}$
computation	$\dfrac{\Gamma, x{:}A \vdash p : \mathbf{Prf}(P)\ \ \Gamma \vdash a : A}{\Gamma \vdash \mathbf{App}(\Lambda x.p, a) = [a/x]p : \mathbf{Prf}([a/x]P)}$	
filling-up	$\dfrac{x \notin FV(p)}{\Gamma \vdash p = \Lambda x.\mathbf{App}(p,x) : \mathbf{Prf}(\forall x{:}A.P)}$	

Figure 1: Rules for proof types.

Figure 1 lists the rules for the proof types, where the following convention on term abbreviation are adopted (similar conventions will be tacitly used for the other type constructors introduced later).

Convention We shall often omit the type information in terms, *e.g.*, to write $\Lambda x.p$ for $\Lambda[\forall x{:}A.P]x.p$. Also, in equality rules, the premises concerning the omitted type information are also omitted. For example, the introduction equality rule in Figure 1 abbreviates the following one:

$$\frac{\Gamma \vdash A = A'\ \ \Gamma, x{:}A \vdash P = P' : \mathit{Prop}\ \ \Gamma, x{:}A \vdash p = p' : \mathbf{Prf}(P)}{\Gamma \vdash \Lambda[\forall x{:}A.P]x.p = \Lambda[\forall x{:}A'.P']x.p' : \mathbf{Prf}(\forall x{:}A.P)}$$

We may write $P' \supset P$ for $\forall x{:}\mathbf{Prf}(P').P$, when $x \notin FV(P)$. $P \supset Q$ and $\forall x{:}A.P$ stand for implication and universal quantification, respectively. The other logical constants and operators can be defined as follows:

$$\begin{aligned}
\mathbf{true} &=_{\mathrm{df}} \forall X{:}\mathit{Prop}.\ X \supset X \\
\mathbf{false} &=_{\mathrm{df}} \forall X{:}\mathit{Prop}.X \\
P_1\ \&\ P_2 &=_{\mathrm{df}} \forall X{:}\mathit{Prop}.\ (P_1 \supset P_2 \supset X) \supset X \\
P_1 \vee P_2 &=_{\mathrm{df}} \forall X{:}\mathit{Prop}.\ (P_1 \supset X) \supset (P_2 \supset X) \supset X \\
\neg P_1 &=_{\mathrm{df}} P_1 \supset \mathbf{false} \\
\exists x \in A.P(x) &=_{\mathrm{df}} \forall X{:}\mathit{Prop}.\ (\forall x{:}A.(P(x) \supset X)) \supset X \\
a =_A b &=_{\mathrm{df}} \forall P{:}A \to \mathit{Prop}.\ P(a) \supset P(b)
\end{aligned}$$

where, $=_A$ between two objects of a type A is called *Leibniz's equality*, which is defined by quantification over the predicates over A. (the objects of the product type $A \to \mathit{Prop}$, see section 5.1). Provability of a proposition P is given by the inhabitance of its proof type $\mathbf{Prf}(P)$; that is, a proposition P is *provable* (in Γ) if $\Gamma \vdash p : \mathbf{Prf}(P)$ for some p.

formation	$\dfrac{\Gamma \vdash A \text{ type } \Gamma, x{:}A \vdash B \text{ type}}{\Gamma \vdash \Pi x{:}A.B \text{ type}}$	$\dfrac{\Gamma \vdash A = A' \ \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash \Pi x{:}A.B = \Pi x{:}A'.B'}$
introduction	$\dfrac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda[\Pi x{:}A.B]x.b : \Pi x{:}A.B}$	$\dfrac{\Gamma, x{:}A \vdash b = b' : B}{\Gamma \vdash \lambda x.b = \lambda x.b' : \Pi x{:}A.B}$
elimination	$\dfrac{\Gamma \vdash f : \Pi x{:}A.B \ \Gamma \vdash a : A}{\Gamma \vdash \mathbf{app}[\Pi x{:}A.B](f,a) : [a/x]B}$	$\dfrac{\Gamma \vdash f = f' : \Pi x{:}A.B \ \Gamma \vdash a = a' : A}{\Gamma \vdash \mathbf{app}(f,a) = \mathbf{app}(f',a') : [a/x]B}$
computation	$\dfrac{\Gamma, x{:}A \vdash b : B \ \Gamma \vdash a : A}{\Gamma \vdash \mathbf{app}(\lambda x.b, a) = [a/x]b : [a/x]B}$	
filling-up	$\dfrac{x \notin FV(f)}{\Gamma \vdash f = \lambda x.\mathbf{app}(f,x) : \Pi x{:}A.B}$	

Figure 2: Rules for Π-types.

5 Type Constructors

5.1 Product types (Π-types)

The rules for product types are listed in Figure 2.

Notation We may write $A \to B$ for $\Pi x{:}A.B$, when $x \notin FV(B)$. We often write $fa_1...a_n$ or $f(a_1,...,a_n)$ for $\mathbf{app}(\mathbf{app}(...\mathbf{app}(f,a_1),...,a_{n-1}),a_n)$. We may write $\lambda x{:}A.b$ to abbreviate $\lambda[\Pi x{:}A.B]x.b$ when hoping to make the domain of a function explicit.

Notation (function composition) We shall use $\circ$ to denote the following composition operator of functions: for f of type $\Pi x_1{:}X_1...\Pi x_n{:}X_n.Y$ with $x_{i_1},...,x_{i_k} \in FV(Y)$ and g of type $\Pi x_{i_1}{:}X_{i_1}...\Pi x_{i_k}{:}X_{i_k}\Pi y{:}Y.Z$,

$$g \circ f =_{\text{df}} \lambda x_1...\lambda x_n.\ g(x_{i_1},...,x_{i_k})(f(x_1,...,x_n))$$

which is of type $\Pi x_1{:}X_1...\Pi x_n{:}X_n.\ [f(x_1,...,x_n)/y]Z$.

5.2 Strong sum (Σ-types)

The rules for Σ-types are listed in Figure 3. We may write $A \times B$ for $\Sigma x{:}A.B$, when $x \notin FV(B)$. The two projection operators can be defined from $\mathbf{E}$ as follows:

$$\pi(c) =_{\text{df}} \mathbf{E}(\lambda x\lambda y.x)(c) \quad \text{and} \quad \pi'(c) =_{\text{df}} \mathbf{E}(\lambda x\lambda y.y)(c)$$

With the filling-up rule, we have, for any f of type $\Pi z{:}(\Sigma x{:}A.B).C$,

$$f = \mathbf{E}(f \circ pair) : \Pi z{:}(\Sigma x{:}A.B).C$$

where $pair =_{\text{df}} \lambda x\lambda y.\mathbf{p}(x,y)$. Also note that the following proposition is provable *without* using the filling-up rules:

$$\forall z{:}(\Sigma x{:}A.B).\ z =_{\Sigma x:A.B} \mathbf{p}(\pi(z),\pi'(z))$$

formation	$\dfrac{\Gamma \vdash A\ \mathbf{type}\ \ \Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash \Sigma x{:}A.B\ \mathbf{type}}$	$\dfrac{\Gamma \vdash A = A'\ \ \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash \Sigma x{:}A.B = \Sigma x{:}A'.B'}$
introduction	$\dfrac{\Gamma \vdash a : A\ \ \Gamma \vdash b : [a/x]B}{\Gamma \vdash \mathbf{p}[\Sigma x{:}A.B](a,b) : \Sigma x{:}A.B}$	$\dfrac{\Gamma \vdash a = a' : A\ \ \Gamma \vdash b = b' : [a/x]B}{\Gamma \vdash \mathbf{p}(a,b) = \mathbf{p}(a',b') : \Sigma x{:}A.B}$
elimination	$\dfrac{\Gamma \vdash d : \Pi x{:}A \Pi y{:}B.[\mathbf{p}(x,y)/z]C}{\Gamma \vdash \mathbf{E}[\Pi z{:}(\Sigma x{:}A.B).C](d) : \Pi z{:}(\Sigma x{:}A.B).C}$	$\dfrac{\Gamma \vdash d = d' : \Pi x{:}A \Pi y{:}B.[\mathbf{p}(x,y)/z]C}{\Gamma \vdash \mathbf{E}(d) = \mathbf{E}(d') : \Pi z{:}(\Sigma x{:}A.B).C}$
computation	$\dfrac{\Gamma \vdash a : A\ \ \Gamma \vdash b : [a/x]B\ \ \Gamma, z{:}\Sigma x{:}A.B \vdash C\ \mathbf{type}\ \ \Gamma \vdash d : \Pi x{:}A \Pi y{:}B.[\mathbf{p}(x,y)/z]C}{\Gamma \vdash \mathbf{E}(d)(\mathbf{p}(a,b)) = d(a,b) : [\mathbf{p}(a,b)/z]C}$	
filling-up	$\dfrac{}{\Gamma \vdash c = \mathbf{p}(\mathbf{E}(\lambda x \lambda y.x)(c), \mathbf{E}(\lambda x \lambda y.y)(c)) : \Sigma x{:}A.B}$	

Figure 3: Rules for Σ-types.

formation	introduction	filling-up
$\dfrac{}{\Gamma \vdash \mathbf{1}\ \mathbf{type}}$	$\dfrac{}{\Gamma \vdash * : \mathbf{1}}$	$\dfrac{}{\Gamma \vdash a = * : \mathbf{1}}$

Figure 4: Rules for the unit type.

5.3 The unit type (1)

The rules for the unit type are given in Figure 4.

Notation (elimination operator for 1) We can define the elimination operator for the unit type as follows: for c of type $[*/z]C$,

$$\mathbf{u}[\Pi z{:}\mathbf{1}.C](c) =_{\mathrm{df}} \lambda[\Pi z{:}\mathbf{1}.C]x.c : \Pi z{:}\mathbf{1}.C$$

It is easy to verify that $\mathbf{u}$ has the following properties (*i.e.*, the following are derivable rules):

$$\frac{\Gamma, z{:}\mathbf{1} \vdash C\ \mathbf{type}\ \ \Gamma \vdash c : [*/z]C}{\Gamma \vdash \mathbf{u}(c)(*) = c : [*/z]C} \qquad \frac{\Gamma \vdash f : \Pi z{:}\mathbf{1}.C}{\Gamma \vdash f = \mathbf{u}(f(*)) : \Pi z{:}\mathbf{1}.C}$$

5.4 Disjoint sum

The rules for disjoint sum types are given in Figure 5.

Remark Note that there is no way to express in a pure equational language that any object of a disjoint sum type is either equal to $\mathbf{i}(a)$ for some a or equal to $\mathbf{j}(b)$ for some b. But we have a (weaker) fact that the proposition

$$\forall z : A + B.\ (\exists x{:}A.z =_{A+B} \mathbf{i}(x) \vee \exists y{:}B.z =_{A+B} \mathbf{j}(y))$$

formation	$\dfrac{\Gamma \vdash A\ \mathbf{type}\ \ \Gamma \vdash B\ \mathbf{type}}{\Gamma \vdash A + B\ \mathbf{type}}$	$\dfrac{\Gamma \vdash A = A'\ \ \Gamma \vdash B = B'}{\Gamma \vdash A + B = A' + B'}$
introduction	$\dfrac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{i}[A+B](a) : A + B}$	$\dfrac{\Gamma \vdash a = a' : A}{\Gamma \vdash \mathbf{i}(a) = \mathbf{i}(a') : A + B}$
	$\dfrac{\Gamma \vdash b : B}{\Gamma \vdash \mathbf{j}[A+B](b) : A + B}$	$\dfrac{\Gamma \vdash b = b' : B}{\Gamma \vdash \mathbf{j}(b) = \mathbf{j}(b') : A + B}$
elimination	$\dfrac{\Gamma \vdash d : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash e : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{case}[\Pi z{:}A+B.C](d,e) : \Pi z{:}A+B.C}$	$\dfrac{\Gamma \vdash d = d' : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash e = e' : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{case}(d,e) = \mathbf{case}(d',e') : \Pi z{:}A+B.C}$
computation	$\dfrac{\Gamma \vdash a : A \quad \Gamma, z{:}A+B \vdash C\ \mathbf{type} \quad \Gamma \vdash d : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash e : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{case}(d,e)(\mathbf{i}(a)) = d(a) : [\mathbf{i}(a)/z]C}$	
	$\dfrac{\Gamma \vdash b : B \quad \Gamma, z{:}A+B \vdash C\ \mathbf{type} \quad \Gamma \vdash d : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash e : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{case}(d,e)(\mathbf{j}(b)) = e(b) : [\mathbf{j}(b)/z]C}$	
filling-up	$\dfrac{}{\Gamma \vdash f = \mathbf{case}(\lambda x.f(\mathbf{i}(x)), \lambda y.f(\mathbf{j}(y))) : \Pi z{:}A+B.C}$	

Figure 5: Rules for disjoint sum types.

formation	elimination	filling-up
$\dfrac{}{\Gamma \vdash \emptyset\ \mathbf{type}}$	$\dfrac{}{\Gamma \vdash \epsilon[\Pi z{:}\emptyset.C] : \Pi z{:}\emptyset.C}$	$\dfrac{}{\Gamma \vdash f = \epsilon : \Pi z{:}\emptyset.C}$

Figure 6: Rules for the empty type.

is provable *without* using the filling-up rules, where $=_{A+B}$ is the propositional (Leibniz's) equality.

5.4.1 finite types

Non-empty finite types can be defined by means of the unit type and the disjoint sum. The finite types N_n $(n > 0)$ can be defined as

$$N_1 =_{\mathrm{df}} \mathbf{1} \qquad N_{n+1} =_{\mathrm{df}} \mathbf{1} + N_n$$

with the n canonical objects of N_n being defined by means of $*$, $\mathbf{i}$ and $\mathbf{j}$.

5.5 The empty type

The rules for the empty type $\emptyset$ are given in Figure 6. Note that there is no introduction rule for it and the filling-up rule gives the uniqueness of functions with the empty type as domain.

formation	$\dfrac{\Gamma \vdash A\ \mathbf{type}\ \ \Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash Wx{:}A.B\ \mathbf{type}}$	$\dfrac{\Gamma \vdash A = A'\ \ \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash Wx{:}A.B = Wx{:}A'.B'}$
introduction	$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B \to Wx{:}A.B}{\Gamma \vdash \mathbf{sup}[Wx{:}A.B](a,b) : Wx{:}A.B}$	$\dfrac{\Gamma \vdash a = a' : A \quad \Gamma \vdash b = b' : [a/x]B \to Wx{:}A.B}{\Gamma \vdash \mathbf{sup}(a,b) = \mathbf{sup}(a',b') : Wx{:}A.B}$
elimination	$\dfrac{\Gamma \vdash e : \Pi x{:}A \Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C}{\Gamma \vdash \mathbf{rec}[\Pi z{:}(Wx{:}A.B).C](e) : \Pi z{:}(Wx{:}A.B).C}$	
	$\dfrac{\Gamma \vdash e = e' : \Pi x{:}A \Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C}{\Gamma \vdash \mathbf{rec}(e) = \mathbf{rec}(e') : \Pi z{:}(Wx{:}A.B).C}$	
computation	$\dfrac{\Gamma \vdash a : A \ \ \Gamma \vdash b : [a/x]B \to Wx{:}A.B \ \ \Gamma, z{:}Wx{:}A.B \vdash C\ \mathbf{type} \quad \Gamma \vdash e : \Pi x{:}A \Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C}{\Gamma \vdash \mathbf{rec}(e)(\mathbf{sup}(a,b)) = e(a,b,\mathbf{rec}(e) \circ b) : [\mathbf{sup}(a,b)/z]C}$	
filling-up	$\dfrac{}{\Gamma \vdash c = \mathbf{sup}(\bar{c},\tilde{c}) : Wx{:}A.B}$	

Figure 7: Rules for W-types.

5.6 Well-ordering (W-types)

W-types [ML84] can be used to define many useful inductive data types. The rules for W-types are given in Figure 7, where, in the filling-up rule,

$$\bar{c} =_{\mathrm{df}} \mathbf{rec}(\lambda x \lambda y \lambda z.x)(c) \quad \tilde{c} =_{\mathrm{df}} \mathbf{rec}(\lambda x \lambda y \lambda z.y)(c)$$

Note that, for $c = \mathbf{sup}(a,b)$, we have $\bar{c} = a$ and $\tilde{c} = b$.

Note that, for any f of type $\Pi z{:}(Wx{:}A.B).C$, we have

$$f = \mathbf{rec}(f \circ sup)$$

where $sup =_{\mathrm{df}} \lambda x \lambda y \lambda z.\mathbf{sup}(x,y)$. Also note that the following proposition is provable *without* using the filling-up rules:

$$\forall z{:}(Wx{:}A.B).\ z =_{Wx:A.B} \mathbf{sup}(\bar{z},\tilde{z})$$

Various inductive data types (and parameterized ones) can be defined as W-types, especially with the help of universes. (See section 7.) A remark is that this is possible only because we have the filling-up rules for various type constructors.

6 Predicative Universes

A universe is a type each of whose objects is a name of some type. Usually, we introduce a universe to reflect the types which have already been defined, in any stage of the development of the theory; in other words, the universe contains the names of the

$\emptyset$	$\dfrac{}{\Gamma \vdash \emptyset : Type_0}$	$\dfrac{}{\Gamma \vdash \mathbf{T}_0(\emptyset) = \emptyset}$
$\mathbf{1}$	$\dfrac{}{\Gamma \vdash \mathbf{1} : Type_0}$	$\dfrac{}{\Gamma \vdash \mathbf{T}_0(\mathbf{1}) = \mathbf{1}}$
$+$	$\dfrac{\Gamma \vdash a : Type_i \quad \Gamma \vdash b : Type_i}{\Gamma \vdash a \oplus_i b : Type_i}$	$\dfrac{\Gamma \vdash a : Type_i \quad \Gamma \vdash b : Type_i}{\Gamma \vdash \mathbf{T}_i(a \oplus_i b) = \mathbf{T}_i(a) + \mathbf{T}_i(b)}$
Φ	$\dfrac{\Gamma \vdash a : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b : Type_i}{\Gamma \vdash \phi_i x{:}a.b : Type_i}$	$\dfrac{\Gamma \vdash a : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b : Type_i}{\Gamma \vdash \mathbf{T}_i(\phi_i x{:}a.b) = \Phi x{:}\mathbf{T}_i(a).\mathbf{T}_i(b)}$
$Type_i$	$\dfrac{}{\Gamma \vdash type_i : Type_{i+1}}$	$\dfrac{}{\Gamma \vdash \mathbf{T}_{i+1}(type_i) = Type_i}$
$\mathbf{T}_i$	$\dfrac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{t}_{i+1}(a) : Type_{i+1}}$	$\dfrac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a)}$
$Prop$	$\dfrac{}{\Gamma \vdash prop : Type_0}$	$\dfrac{}{\Gamma \vdash \mathbf{T}_0(prop) = Prop}$
$\mathbf{Prf}$	$\dfrac{\Gamma \vdash P : Prop}{\Gamma \vdash \mathbf{t}_0(P) : Type_0}$	$\dfrac{\Gamma \vdash P : Prop}{\Gamma \vdash \mathbf{T}_0(\mathbf{t}_0(P)) = \mathbf{Prf}(P)}$

Figure 8: Universe introduction and reflection rules (Φ/ϕ_i for Π/π_i, Σ/σ_i and W/w_i).

types defined in an earlier stage. Introducing such a universe is *predicative* in the sense that only the existing types are reflected, but not types which are introduced after the universe; in particular, the universe does not reflect itself — it does not contain a name for itself. After introducing a universe, we can then add another (stronger) universe to reflect the defined types, including the previous universe. This development can continue to introduce infinite many universes, as we show below.

We introduce predicative universes $Type_i$ ($i \in \omega$), which have the following formation rules:

$$\frac{}{\Gamma \vdash Type_i\ \mathbf{type}} \qquad \frac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{T}_i(a)\ \mathbf{type}} \qquad \frac{\Gamma \vdash a = a' : Type_i}{\Gamma \vdash \mathbf{T}_i(a) = \mathbf{T}_i(a')}$$

The introduction and reflection rules are given in Figure 8, to which we should also add the following introduction rules for object equality:

$$\frac{\Gamma \vdash a = a' : Type_i \quad \Gamma \vdash b = b' : Type_i}{\Gamma \vdash a \oplus_i b = a' \oplus_i b' : Type_i} \qquad \frac{\Gamma \vdash a = a' : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b = b' : Type_i}{\Gamma \vdash \phi_i x{:}a.b = \phi_i x{:}a'.b' : Type_i}$$

$$\frac{\Gamma \vdash a = a' : Type_i}{\Gamma \vdash \mathbf{t}_{i+1}(a) = \mathbf{t}_{i+1}(a') : Type_{i+1}} \qquad \frac{\Gamma \vdash P = P' : Prop}{\Gamma \vdash \mathbf{t}_0(P) = \mathbf{t}_0(P') : Type_0}$$

We may also include the following reflection rules which have the effect that type equalities are also reflected by universes (and the names of types are unique):

$$\frac{\Gamma \vdash a : Type_i \quad \Gamma \vdash b : Type_i}{\Gamma \vdash \mathbf{t}_{i+1}(a \oplus_i b) = \mathbf{t}_{i+1}(a) \oplus_{i+1} \mathbf{t}_{i+1}(b) : Type_{i+1}}$$

$$\frac{\Gamma \vdash a : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b : Type_i}{\Gamma \vdash \mathbf{t}_{i+1}(\phi_i x{:}a.b) = \phi_{i+1} x{:}\mathbf{t}_{i+1}(a).\mathbf{t}_{i+1}(b) : Type_{i+1}}$$

Remark The system is supposed to be *open* (*c.f.*, Martin-Löf's type theory). That is, new types (and new type constructors) can be added to the system when it is needed. This is one of the reasons[3] that we have not included universe elimination rules which would impose the principle of primitive recursion on universes and essentially make the universes closed (*e.g.*, to extend the system, one would have to *modify* the universe elimination rules). The reflection principle by predicative universes provides a powerful tool to do abstract reasoning and program development and is enough in practice, especially when typical ambiguity is implemented in proof development systems like Lego (*c.f.*, [Pol90]).

7 Inductive Data Types via. W-types

W-types in the theory can be used to define various inductive data types. A key observation is that the filling-up rules are necessary (and sufficient) for W-types to represent data types faithfully. In other words, it is not necessary to have an extensional propositional equality (*c.f.*, [ML84]) for W-types to serve as a mechanism for defining inductive data types. This is the main motivation to introduce filling-up equality rules in the theory. We give several simple examples to explain this.

Example (the type of natural numbers) The type of natural numbers can be defined as

$$N =_{\mathrm{df}} Wx{:}A_N.B_N$$

with

$$A_N \equiv \mathbf{1} + \mathbf{1} \quad \text{and} \quad B_N \equiv \mathbf{T}_0(\mathbf{case}(\mathbf{u}(\emptyset), \mathbf{u}(1))(x))$$

where $\emptyset$ and 1 are the names of the empty type and the unit type, respectively. The canonical natural numbers are defined as

$$0 =_{\mathrm{df}} \mathbf{sup}(\mathbf{i}(*), \epsilon) \quad \text{and} \quad succ(n) =_{\mathrm{df}} \mathbf{sup}(\mathbf{j}(*), \mathbf{u}(n))$$

Note that, by the filling-up rules for the unit type and the empty type, any natural number (an object of type N) of the form $\mathbf{sup}(\mathbf{i}(a), f)$ is equal to 0.

Now, we use the elimination operator **rec** for W-types to define an elimination operator $\mathbf{rec}_N$ which should satisfy the following elimination rule:

$$\frac{\Gamma, n{:}N \vdash C\ \mathbf{type} \quad \Gamma \vdash c : [0/n]C \quad \Gamma \vdash f : \Pi n{:}N.\ C \to [succ(n)/n]C}{\Gamma \vdash \mathbf{rec}_N(c, f) : \Pi n{:}N.C}$$

[3] Another reason is more technical, that is, primitive recursion over universes is logically inconsistent with the rules that reflect type equalities.

and the equalities $\mathbf{rec}_N(c,f)(0) = c$ and $\mathbf{rec}_N(c,f)(succ(n)) = f(n,\mathbf{rec}_N(c,f)(n))$. $\mathbf{rec}_N$ is defined as follows:

$$\mathbf{rec}_N(c,f) =_{\mathrm{df}} \mathbf{rec}(f')$$

where

$$f' \equiv \mathbf{case}(\mathbf{u}(\lambda y\lambda z.c), \mathbf{u}(\lambda y\lambda z.f(y(*),z(*)))))$$

Note that, for $\mathbf{rec}(f')$ to have the correct type (that is, $\Pi n{:}N.C$), f' must be of type

$$\Pi x{:}A_N\Pi y{:}B_N \to N\Pi z{:}(\Pi v{:}B_N.[y(v)/n]C).\ [\mathbf{sup}(x,y)/n]C$$

This is the case because of the filling-up rules for the empty type, the unit type and the Π-types. To see this, we only have to notice that $\mathbf{u}(\lambda y\lambda z.c)$ is of type

$$\begin{aligned}
& \Pi x'{:}1\Pi y{:}\emptyset \to N\Pi z{:}(\Pi v{:}\emptyset.[y(v)/n]C).\ [0/n]C \\
\equiv\ & \Pi x'{:}1\Pi y{:}\emptyset \to N\Pi z{:}(\Pi v{:}\emptyset.[y(v)/n]C).\ [\mathbf{sup}(\mathbf{i}(*),\epsilon)/n]C \\
=\ & \Pi x'{:}1\Pi y{:}\emptyset \to N\Pi z{:}(\Pi v{:}\emptyset.[y(v)/n]C).\ [\mathbf{sup}(\mathbf{i}(x'),y)/n]C \\
=\ & \Pi x'{:}1\Pi y{:}([\mathbf{i}(x')/x]B_N) \to N\Pi z{:}(\Pi v{:}([\mathbf{i}(x')/x]B_N).[y(v)/n]C).\ [\mathbf{sup}(\mathbf{i}(x'),y)/n]C \\
\equiv\ & \Pi x'{:}1.\ [\mathbf{i}(x')/x]\Pi y{:}B_N \to N\Pi z{:}(\Pi v{:}B_N.[y(v)/n]C).\ [\mathbf{sup}(x,y)/n]C
\end{aligned}$$

and $\mathbf{u}(\lambda y\lambda z.f(y(*),z(*)))$ is of type

$$\begin{aligned}
& \Pi y'{:}1\Pi y{:}1 \to N\Pi z{:}(\Pi v{:}1.[y(v)/n]C).\ [succ(y(*))/n]C \\
\equiv\ & \Pi y'{:}1\Pi y{:}1 \to N\Pi z{:}(\Pi v{:}1.[y(v)/n]C).\ [\mathbf{sup}(\mathbf{j}(*),\mathbf{u}(y(*)))/n]C \\
=\ & \Pi y'{:}1\Pi y{:}1 \to N\Pi z{:}(\Pi v{:}1.[y(v)/n]C).\ [\mathbf{sup}(\mathbf{j}(y'),y)/n]C \\
=\ & \Pi y'{:}1\Pi y{:}([\mathbf{j}(y')/x]B_N) \to N\Pi z{:}(\Pi v{:}([\mathbf{j}(y')/x]B_N).[y(v)/n]C).\ [\mathbf{sup}(\mathbf{j}(y'),y)/n]C \\
\equiv\ & \Pi y'{:}1\ [\mathbf{j}(y')/x]\Pi y{:}B_N \to N\Pi z{:}(\Pi v{:}B_N.[y(v)/n]C).\ [\mathbf{sup}(x,y)/n]C
\end{aligned}$$

It is easy to verify that $\mathbf{rec}_N$ satisfies the required equalities.

Other data types can be defined in a similar way. For example, the type of lists of natural numbers can be defined as follows:

$$List(N) =_{\mathrm{df}} Wx{:}A_{List(N)}.B_{List(N)}$$

with $A_{List(N)} \equiv 1 + N$ and $B_{List(N)} \equiv \mathrm{T}_0(\mathbf{case}(\mathbf{u}(\emptyset),\mathbf{u}(1))(x))$. The 'canonical' objects of $List(N)$ are defined as $nil =_{\mathrm{df}} \mathbf{sup}(\mathbf{i}(*),\epsilon)$ and $cons(n,l) =_{\mathrm{df}} \mathbf{sup}(\mathbf{j}(n),\mathbf{u}(l))$, and the elimination operator as, for $c : [nil/l]C$ and $f : \Pi n{:}N\Pi l{:}List(N).\ C \to [cons(n,l)/l]C$,

$$\mathbf{rec}_{List(N)}(c,f) =_{\mathrm{df}} \mathbf{rec}(\mathbf{case}(\mathbf{u}(\lambda y\lambda z.c), \lambda n\lambda y\lambda z.f(n,y(*),z(*))))$$

which is of type $\Pi l{:}List(N).C$ (we need the same filling-up rules as those used for N to verify this) and satisfies the equalities $\mathbf{rec}_{List(N)}(c,f)(nil) = c$ and $\mathbf{rec}_{List(N)}(c,f)(cons(n,l)) = f(n,l,\mathbf{rec}_{List(N)}(c,f)(l))$.

As another example, the type of binary trees with nodes labeled by natural numbers can be defined as:

$$BT(N) =_{\text{df}} Wx{:}A_{BT(N)}.B_{BT(N)}$$

with $A_{BT(N)} \equiv \mathbf{1} + N$ and $B_{BT(N)} \equiv \mathbf{T}_0(\mathbf{case}(\mathbf{u}(\emptyset), \mathbf{u}(1 \oplus 1))(x))$. The 'canonical' objects of type $BT(N)$ are defined as $empty =_{\text{df}} \mathbf{sup}(\mathbf{i}(*), \epsilon)$ and $mk(n, t_1, t_2) =_{\text{df}} \mathbf{sup}(\mathbf{j}(n), \mathbf{case}(\mathbf{u}(t_1), \mathbf{u}(t_2)))$, and the elimination operator as, for $c : [empty/t]C$ and $f : \Pi n{:}N \Pi t_1, t_2{:}BT(N).\ [t_1/t]C \to [t_2/t]C \to [mk(n, t_1, t_2)/t]C$,

$$\mathbf{rec}_{BT(N)}(c, f) =_{\text{df}} \mathbf{rec}(\mathbf{case}(\mathbf{u}(\lambda y \lambda z.c), \lambda n \lambda y \lambda z.f(n, y(\mathbf{i}(*)), y(\mathbf{j}(*)), z(\mathbf{i}(*)), z(\mathbf{j}(*)))))$$

which has type $\Pi t{:}BT(N).C$. To verify this last fact, we need thr filling-up rule for disjoint union types as well as those used for N and $List(N)$. The elimination operator satisfies the equalities $\mathbf{rec}_{BT(N)}(c, f)(empty) = c$ and $\mathbf{rec}_{BT(N)}(c, f)(mk(n, t_1, t_2)) = f(n, t_1, t_2, \mathbf{rec}_{BT(N)}(c, f)(t_1), \mathbf{rec}_{BT(N)}(c, f)(t_2))$.

Parameterized inductive data types can also be defined by means of universes. For example, for any universe U, we can define

$$list_U =_{\text{df}} \lambda a{:}U.\ wx{:}(1 \oplus a).\mathbf{case}(\mathbf{u}(\emptyset), \mathbf{u}(1))(x)$$

Then, we have $\mathbf{T}_U(list_U(a)) = List(\mathbf{T}_U(a))$, for $a : U$.

8 Subtyping

There is another relationship between types, called cumulativity or subtyping. For types A and B, $A \preceq B$ means that every object of type A is also of type B. We consider subtyping as a partial order between types modulo type equality. Instead of introducing $\preceq$ directly, we introduce the relation $\prec$, the proper subtyping; Intuitively, $A \prec B$ means that every object of type A is also of type B, but A and B are not equal. So, we may consider judgements of the form

$$\Gamma \vdash A \prec B$$

We adopt a similar convention that, to derive the above judgement, $\Gamma \vdash A$ **type** and $\Gamma \vdash B$ **type** must have been derived.

8.1 Subtyping: the general rules

$\prec$ is transitive modulo to type equality; so we have the following rules.

$$\frac{\Gamma \vdash A = B \quad \Gamma \vdash B \prec C \quad \Gamma \vdash C = D}{\Gamma \vdash A \prec D} \qquad \frac{\Gamma \vdash A \prec B \quad \Gamma \vdash B \prec C}{\Gamma \vdash A \prec C}$$

The intuitive semantics of subtyping is reflected by the following rules, which are analogous to the equality typing rules:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \prec B}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a = a' : A \quad \Gamma \vdash A \prec B}{\Gamma \vdash a = a' : B}$$

Π	$\dfrac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B \prec B'}{\Gamma \vdash \Pi x{:}A.B \prec \Pi x{:}A'.B'}$
Σ	$\dfrac{\Gamma \vdash A \prec A' \quad \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash \Sigma x{:}A.B \prec \Sigma x{:}A'.B'} \qquad \dfrac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B \prec B'}{\Gamma \vdash \Sigma x{:}A.B \prec \Sigma x{:}A'.B'}$
$+$	$\dfrac{\Gamma \vdash A \prec A' \quad \Gamma \vdash B = B'}{\Gamma \vdash A + B \prec A' + B'} \qquad \dfrac{\Gamma \vdash A = A' \quad \Gamma \vdash B \prec B'}{\Gamma \vdash A + B \prec A' + B'}$
W	$\dfrac{\Gamma \vdash A \prec A' \quad \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash Wx{:}A.B \prec Wx{:}A'.B'}$

Figure 9: Subtyping rules for type constructors.

With $\prec$ given, we can introduce $\preceq$ by the following rules.

$$\frac{\Gamma \vdash A = B}{\Gamma \vdash A \preceq B} \qquad \frac{\Gamma \vdash A \prec B}{\Gamma \vdash A \preceq B}$$

Remark One might also postulate an antisymmetry rule saying that $A = B$ if $A \preceq B$ and $B \preceq A$. However, we take the above two rules as the only rules which can derive a 'judgement' of the form $\Gamma \vdash A \preceq B$.

8.2 Subtyping rules for type constructors

We have further formation rules for some of the type constructors which tell us how the types constructed by the constructor are related by subtyping. The rules for Π, Σ, $+$ and W are given in Figure 9.

One may also consider to add the following subtyping rules for Π and W, respectively:

$$\frac{\Gamma \vdash A' \prec A \quad \Gamma, x{:}A' \vdash B = B'}{\Gamma \vdash \Pi x{:}A.B \prec \Pi x{:}A'.B'} \qquad \frac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B' \prec B}{\Gamma \vdash Wx{:}A.B \prec Wx{:}A'.B'}$$

which makes the subtyping contravariant for Π and W.

8.3 Cumulativity induced by universe inclusions

Subtyping is often desirable in practice. They give good intuitions and simpler and more concise expressions. A good example in this aspect is to use subtyping to consider universe inclusions. For example, the lifting operators $\mathbf{t}_i$ can be removed by assuming that $Type_i$ is included in $Type_{i+1}$ and $Prop$ is included in $Type_0$. However, if one simply removed $\mathbf{t}_i$ in the rules in the last section, the resulting rules would not capture our intuition very well; they have a problem that the inclusions between universes do not propagate properly to the types constructed by some type constructors. For example, it is easy to show (by induction on derivations) that

$$f{:}Type_0 \to Type_0 \not\vdash f : Type_0 \to Type_1$$

Such a problem disappears if we use subtyping rules to express the universe inclusions, since the subtyping rules for type constructors play the role of subtyping propagation. We have the following introduction rules for inclusions between (predicative) universes, where $i \in \omega$:

$$\frac{}{\Gamma \vdash Type_i \prec Type_{i+1}} \qquad \frac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{T}_{i+1}(a) = \mathbf{T}_i(a)}$$

Lifting propositions and reflecting the proof types, we have rules

$$\frac{}{\Gamma \vdash Prop \prec Type_0} \qquad \frac{\Gamma \vdash P : Prop}{\Gamma \vdash \mathbf{T}_0(P) = \mathbf{Prf}(P)}$$

Of course, with these rules, the rules for reflecting $\mathbf{T}_i$ and **Prf** should be removed.

References

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[Coq89] Th. Coquand. Metamathematical investigations of a calculus of constructions. manuscript, 1989.

[Luo89] Zhaohui Luo. **ECC**, an extended calculus of constructions. In *Proc. of the Fourth Ann. Symp. on Logic in Computer Science*, Asilomar, California, U.S.A., June 1989.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[ML75] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.Rose and J.C.Shepherdson, editors, *Logic Colloquium'73*, 1975.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an introduction.* Oxford University Press, 1990.

[Pol90] R. Pollack. Implicit syntax. In the preliminary Proceedings of the 1st Workshop on Logical Frameworks, 1990.

A A Formulation in Martin-Löf's framework

The following gives a formulation of the type system described in section 2 to section 6 in Martin-Löf's framework [NPS90]. We shall use **Type** instead of **Set** as used in [NPS90]. We also omit the lifting operator *El* to simply write A for $El(A)$.

A.1 The impredicative universe

Introduction of constants:

$$\begin{aligned} \mathit{Prop} &: \mathbf{Type} \\ \mathbf{Prf} &: (\mathit{Prop})\mathbf{Type} \\ \forall &: (A{:}\mathbf{Type})((A)\mathit{Prop})\mathit{Prop} \\ \Lambda &: (A{:}\mathbf{Type})(P{:}(A)\mathit{Prop})((x{:}A)\mathbf{Prf}(P(x)))\mathbf{Prf}(\forall(A,P)) \\ \mathbf{App} &: (A{:}\mathbf{Type})(P{:}(A)\mathit{Prop})(\mathbf{Prf}(\forall(A,P)))(a{:}A)\mathbf{Prf}(P(a)) \end{aligned}$$

Assertion of equalities:

For A:**Type**, P:$(A)\mathit{Prop}$, f:$(x{:}A)\mathbf{Prf}(P(x))$ and a:A,

$$\mathbf{App}(A,P,\Lambda(A,P,f),a) = f(a) : \mathbf{Prf}(P(a))$$

For A:**Type**, P:$(A)\mathit{Prop}$, p:$\mathbf{Prf}(\forall(A,P))$,

$$p = \Lambda(A,P,(x)\mathbf{App}(A,P,p,x)) : \mathbf{Prf}(\forall(A,P))$$

A.2 Product types (Π-types)

Introduction of constants:

$$\begin{aligned} \Pi &: (A{:}\mathbf{Type})((A)\mathbf{Type})\mathbf{Type} \\ \lambda &: (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})((x{:}A)B(x))\Pi(A,B) \\ \mathbf{app} &: (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(\Pi(A,B))(a{:}A)B(a) \end{aligned}$$

Assertion of equalities:

For A:**Type**, B:(A)**Type**, f:$(x{:}A)B(x)$ and a:A,

$$\mathbf{app}(A, B, \lambda(A, B, f), a) = f(a) : B(a)$$

For A:**Type**, B:(A)**Type**, f:$\Pi(A, B)$,

$$f = \lambda(A, B, (x)\mathbf{app}(A, B, f, x)) : \Pi(A, B)$$

A.3 Strong sum (Σ-types)

Introduction of constants:

$$\begin{aligned}
\Sigma &: (A{:}\mathbf{Type})((A)\mathbf{Type})\mathbf{Type} \\
\mathbf{p} &: (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(x{:}A)(B(x))\Sigma(A, B) \\
\mathbf{E} &: (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(C{:}(\Sigma(A, B))\mathbf{Type}) \\
&\quad ((x{:}A)(y{:}B(x))C(\mathbf{p}(A, B, x, y)))(z{:}\Sigma(A, B))C(z)
\end{aligned}$$

Assertion of equalities:

For A:**Type**, B:(A)**Type**, C:$(\Sigma(A, B))$**Type**, f:$(x{:}A)(y{:}B(x))C(\mathbf{p}(A, B, x, y))$, a:A and b:$B(a)$,

$$\mathbf{E}(A, B, C, f, \mathbf{p}(A, B, a, b)) = f(a, b) : C(\mathbf{p}(A, B, a, b))$$

For A:**Type**, B:(A)**Type** and c:$\Sigma(A, B)$,

$$c = \mathbf{p}(A, B, \pi(c), \pi'(c)) : B(\pi(c))$$

where $\pi(c) \equiv \mathbf{E}(A, B, (z)A, (x)(y)x, c)$ and $\pi'(c) \equiv \mathbf{E}(A, B, (z)B(\pi(c)), (x)(y)y, c)$, for notational simplicity.

A.4 The unit type

Introduction of constants:

$$\begin{aligned}
\mathbf{1} &: \mathbf{Type} \\
* &: \mathbf{1}
\end{aligned}$$

Assertion of equalities:

For a:**1**,

$$a = * : \mathbf{1}$$

A.5 Disjoint sum

Introduction of constants:

$$\begin{aligned} + &: (\mathbf{Type})(\mathbf{Type})\mathbf{Type} \\ \mathbf{i} &: (A, B{:}\mathbf{Type})(A) + (A, B) \\ \mathbf{j} &: (A, B{:}\mathbf{Type})(B) + (A, B) \\ \mathbf{case} &: (A, B{:}\mathbf{Type})(C{:}(+(A, B))\mathbf{Type}) \\ &\quad ((x{:}A)C(\mathbf{i}(A, B, x)))((y{:}B)C(\mathbf{j}(A, B, y)))(z{:}+(A, B))C(z) \end{aligned}$$

Assertion of equalities:

For A, B:**Type**, C:$(+(A, B))$**Type**, f:$(x{:}A)C(\mathbf{i}(A, B, x))$, g:$(y{:}B)C(\mathbf{j}(A, B, y))$, a:A and b:B,

$$\mathbf{case}(A, B, C, f, g, \mathbf{i}(A, B, a)) = f(a) : C(\mathbf{i}(A, B, a))$$

$$\mathbf{case}(A, B, C, f, g, \mathbf{j}(A, B, b)) = g(b) : C(\mathbf{j}(A, B, b))$$

For A, B:**Type**, C:$(+(A, B))$**Type**, f:$(z{:}+(A, B))C(z)$,

$$f = \mathbf{case}(A, B, C, (x)f(\mathbf{i}(A, B, x)), (y)f(\mathbf{j}(A, B, y))) : (z{:}+(A, B))C(z)$$

A.6 The empty type

Introduction of constants:

$$\begin{aligned} \emptyset &: \mathbf{Type} \\ \epsilon &: (C{:}(\emptyset)\mathbf{Type})(z{:}\emptyset)C(z) \end{aligned}$$

Assertion of equalities:

For C:$(\emptyset)$**Type** and f:$(z$:$\emptyset)C(z)$,

$$f = \epsilon(C) : (z{:}\emptyset)C(z)$$

A.7 Well-ordering (W-types)

Introduction of constants:

$$\begin{aligned}
W &: (A{:}\mathbf{Type})((A)\mathbf{Type})\mathbf{Type} \\
\mathbf{sup} &: (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(x{:}A)((B(x))W(A,B))W(A,B) \\
\mathbf{rec} &: (A{:}\mathbf{Type})(B{:}(A)\mathbf{Type})(C{:}(W(A,B))\mathbf{Type}) \\
&\quad ((x{:}A)(y{:}(B(x))W(A,B))((v{:}B(x))C(y(v)))C(\mathbf{sup}(A,B,x,y))) \\
&\quad (z{:}W(A,B))C(z)
\end{aligned}$$

Assertion of equalities:

For A:**Type**, B:(A)**Type**, C:$(W(A,B))$**Type**, a:A, b:$(B(a))W(A,B)$, and f:$(x{:}A)(y{:}(B(x))W(A,B))((v{:}B(x))C(y(v)))C(\mathbf{sup}(A,B,x,y))$,

$$\mathbf{rec}(A,B,C,f,\mathbf{sup}(A,B,a,b)) = f(a,b,(v)\mathbf{rec}(A,B,C,f,b(v))) : C(\mathbf{sup}(A,B,a,b))$$

For A:**Type**, B:(A)**Type** and c:$W(A,B)$,

$$c = \mathbf{sup}(A,B,\bar{c},\tilde{c}) : W(A,B)$$

where $\bar{c} \equiv \mathbf{rec}(A,B,(z)A,(x)(y)(v)x,c)$ and $\tilde{c} \equiv \mathbf{rec}(A,B,(z)(B(\bar{c}))W(A,B),(x)(y)(v)y,c)$, for notational simplicity.

A.8 Predicative universes

In the following, $i \in \omega$ and $\phi_i \in \{\pi_i, \sigma_i, w_i\}$.

Introduction of constants:

$$\begin{aligned}
Type_i &: \mathbf{Type} \\
\mathbf{T}_i &: (Type_i)\mathbf{Type} \\
\emptyset &: Type_0 \\
1 &: Type_0 \\
\oplus_i &: (Type_i)(Type_i)Type_i \\
\phi_i &: (x{:}Type_i)((\mathbf{T}_i(x))Type_i)Type_i \\
type_i &: Type_{i+1} \\
\mathbf{t}_{i+1} &: (Type_i)Type_{i+1} \\
prop &: Type_0 \\
\mathbf{t}_0 &: (Prop)Type_0
\end{aligned}$$

Assertion of equalities:

$$\mathbf{T}_0(\emptyset) = \emptyset$$

$$\mathbf{T}_0(1) = \mathbf{1}$$

For $a, b{:}Type_i$,

$$\mathbf{T}_i(\oplus_i(a,b)) = +(\mathbf{T}_i(a), \mathbf{T}_i(b))$$

For $a{:}Type_i$ and $b{:}(\mathbf{T}_i(a))Type_i$,

$$\mathbf{T}_i(\phi_i(a,b)) = \Phi(\mathbf{T}_i(a), (x)\mathbf{T}_i(b(x)))$$

where $\Phi \equiv \Pi,\ \Sigma,\ W$ for $\phi_i \equiv \pi_i, \sigma_i, w_i$, respectively.

$$\mathbf{T}_{i+1}(type_i) = Type_i$$

For $a{:}Type_i$,

$$\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a)$$

$$\mathbf{T}_0(prop) = Prop$$

For $P{:}Prop$,

$$\mathbf{T}_0(\mathbf{t}_0(P)) = \mathbf{Prf}(P)$$

For $a, b{:}Type_i$,

$$\mathbf{t}_{i+1}(\oplus_i(a,b)) = \oplus_{i+1}(\mathbf{t}_{i+1}(a), \mathbf{t}_{i+1}(b)) : Type_{i+1}$$

For $a{:}Type_i$ and $b{:}(\mathbf{T}_i(a))Type_i$,

$$\mathbf{t}_{i+1}(\phi_i(a,b)) = \phi_{i+1}(\mathbf{t}_{i+1}(a), (x)\mathbf{t}_{i+1}(b(x))) : Type_{i+1}$$