A Unifying Theory of Dependent Types: the Schematic Approach

Zhaohui Luo
Department of Computer Science, University of Edinburgh
JCMB, KB, Mayfield Rd., Edinburgh EH9 3JZ, U.K.

1 Introduction

We present a theory of dependent types which unifies coherently Martin-Löf's type theory with universes [ML75, ML84, NPS90] and Coquand-Huet's Calculus of Constructions [CH88, Coq89]. The theory can be seen as an extension of the Extended Calculus of Constructions [Luo89, Luo90a] by a large class of inductive data types. It is a further development of the idea to enhance in type theory a conceptual distinction between the notions of logical formula (proposition) and data type, and is another step aiming at the development of a unifying language for modular development of programs, specifications, and proofs (*c.f.*, [Luo91b]).

The presentation here is particularly inspired by Martin-Löf's presentation of type theory by logical framework [NPS90] and Coquand and Mohring's work on inductive types [CPM90]. The type theory is formulated in a logical framework extended by kind schemata. It consists of an impredicative universe of logical propositions, a class of inductive data types covered by a general form of schemata, and predicative type universes. Our presentation is different from that of [CPM90] in the following aspects. Using a logical framework allows us to define a purely intensional type theory with a general schema for inductive types. In particular, besides distinguishing the notions of logical proposition and data type, we pay special attention to the intensionality of computational equality and how to obtain a clearly non-circular reflection principle in introducing predicative universes. Furthermore, our approach is more moderate in that, for example, our schemata only cover inductive types but not inductive relations. There are two reasons in favor of such a moderate approach. First, it seems that the necessity to cover general inductive relations by the schemata is still to be justified; in our setting, logical inductive relations can be obtained by impredicative definitions and many inductive families of data types can be defined using predicative universes. Second, by taking such a moderate approach, we hope that the resulting type theory allows a molecular or compositional understanding, as we shall briefly explain in the conclusion.

Several basic ideas are elaborated below to explain our motivations and points of view.

1.1 Data types vs. logical propositions

The Curry-Howard correspondence between propositions and types has been the basic key idea in the development of various type theories. However, although propositions can be viewed as types, identifying types with propositions does not seem to be necessary or satisfactory. In other words, logical propositions can be viewed as types of their proofs, but it is not necessary that every type be viewed as a logical proposition. It is our view that, even in type theory, a distinction between the notions of logical formula and data type is both conceptually natural and pragmatically important. This idea has been reflected, for example, in the development of the Extended Calculus of Constructions (**ECC**) [Luo89, Luo90a], where higher-order logical propositions reside in the impredicative universe (*c.f.*, the calculus of constructions [CH88]), while the data types (or sets) reside in the predicative universes (see Figure 1).

Intuitively, such a type theory may be understood by considering its *conceptual universe of types*, which reflects the way we try to understand the real world of objects of interest. To grasp this conceptual universe, we pick out types and type constructors to study, examples of which include the type of natural numbers,

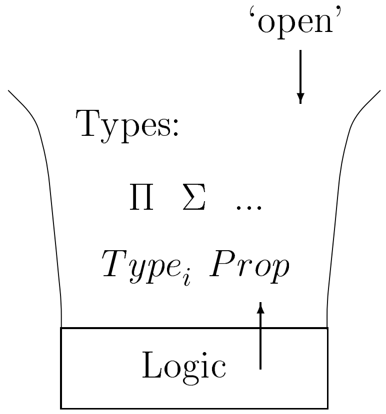

Figure 1: The conceptual universe of types in **ECC**.

the dependent product types, the strong sum types, *etc.* In particular, one may consider a type of logical propositions which are themselves (names of) types of their proofs and provide the means to describe the logical properties of objects of any type. One can also introduce (formal) type universes in a stratified or predicative way, which consist of names of the types already introduced into the theory. The predicativity of a universe means that the existence of the data types that have names in it should be independent on the existence of the universe itself. Since we do not expect to exhaust the possible conceptions of type formation, the conceptual universe of types is supposed to be open. However, it seems possible and reasonable to consider the (impredicative) world of logical propositions as relatively closed, in the sense that the ways (*e.g.*, in the case of higher-order logic, the universal quantification) in which logical propositions are formed are not supposed to be further extended. Therefore, the openness essentially concerns the predicative part of the theory, where the objects like programs and data types reside.

The general structure of the conceptual universe of types as sketched above provides a particular but coherent view which seems to be of importance both in pragmatic applications and theoretical study. For example, since the logical universe is relatively closed, there is an internal notion of predicate (as propositional function) in the theory which is useful in many applications (*e.g.*, in developing an approach to program specification and development [BM91][Luo91b] and a notion of mathematical theory in the application of abstract reasoning [Luo91a][Pol90]). Another evidence supporting this view is that, since data types (sets) reside in predicative universes, the embedded logic (in **ECC**, for example) is a conservative extension of the higher-order predicate logic [Luo90b], and we are also able to grasp and use type constructors like the strong sum (large Σ-types), which is useful in abstract and modular development of programs and proofs. It is also our hope that, although the logical universe of propositions is impredicative, a hierarchical understanding of the language of the type theory can be obtained in certain reasonable sense (see section 5.1).

1.2 The philosophical openness vs. pragmatic considerations

The openness of the conceptual universe of types, as well-explained by Martin-Löf for his type theory, is philosophically sound and satisfactory. On the other hand, when studying a formal type theory, one has to 'close' the system at some stage of its development in order to, *e.g.*, study its meta theory or implement it on computer. Such a consideration encourages people to consider various general formulations of classes of types in order to deal with them in a uniform way. In fact, such a generality is not a new subject. It is analogous to, for example, the introduction of all of the dependent product types by a schematic type constructor (Π) instead of introducing them one by one. The issue of introducing a large class of inductive data types has been considered by, for example, Backhouse [Bac88] and Dybjer [Dyb89, Dyb91] for Martin-Löf's type theory, and Coquand and Mohring [CPM90], Ore [Ore90] and Goguen and Luo [GL91] for impredicative systems. A useful idea, which goes back to Gentzen and is developed by Prawitz and Martin-Löf among others, is that the meaning of an inductive type may be viewed as determined by its introduction rules. This idea has in particular been developed by Backhouse, Dybier, Coquand and Mohring in the context of type theory to consider general schemata for inductive types. The work reported here is partly along this line of research.

1.3 Intensionality vs. extensionality

From a computational point of view, we think that the computational equality between objects of types (say, natural numbers, functions of Π-types) should be intensional. By this, we mean that neither the strong extensionality (for example, formulated by the strong equality types in [ML84]) nor the weak extensionality as expressed by the so-called filling-up rules expressing the uniqueness of the elimination operator in a type theory[1] should be viewed as computational.

However, in most of the current formulations of inductive data types, certain extensional equalities have been used in order to get satisfactory representation of inductive types. For example, in [CPM90], η-rule for the dependent product types is included in order to gain a satisfactory formulation of the general schema for inductive types. In [GL91], various filling-up rules are studied and shown to be sufficient and necessary for the well-ordering types to be used to represent various inductive types faithfully. It seems to be the case that certain extensionality is called for in order to cover a large class of inductive data types by a general representation schema.

A solution to such a dilemma is to separate the type theory (object language to be defined) from a meta-language used to define it. This latter meta-language can (and should) be weakly extensional. This seems to be exactly Martin-Löf's idea of using a logical framework with η-rule to define his type theory (see [NPS90], also *c.f.*, [HHP87]). As shown below, based on such an idea, we can consider a notion of kind schemata at the level of framework which can be used to define a large class of inductive types, including the Π-types, which are intensional in the sense that no extensional equality rule between objects of types holds at the computational level. Furthermore, the filling-up equality rules like η do hold logically in the sense that one can prove the corresponding logical proposition by induction.

Another advantage of using a meta-level logical framework is that we gain a clear reflection principle in the sense that we do not need to use a predicative type universe to help the formulation of inductive types residing in it, which, otherwise, would give a flavor of impredicativity. In our setting, inductive types are introduced independently and predicative universes are viewed as introduced later by declaring names of certain data types in it.

In section 2, we first describe Martin-Löf's logical framework LF, where as an example, we formulate the impredicative universe in the type theory, and then introduce LF_θ, a simple extension of LF by schemata. The general rules for inductive data types are introduced in section 3, where we give examples and discuss intensionality. Predicative universes and the reflection of inductive types are introduced in section 4, followed by section 5 giving a summary of the type theory and discussing some related topics.

2 Logical Framework with Schemata

2.1 Martin-Löf's logical framework

We consider LF, a typed version of Martin-Löf's logical framework [NPS90]. The inference rules of LF are given in Appendix A. For those who are familiar with the presentation in [NPS90], please notice the following notational changes:

1. We call the types in LF *kinds* and write 'K **kind**' (instead of 'K **type**').

2. The kind of all types is denoted by **Type** (instead of **Set**). We shall often omit the lifting operator El.

3. We shall write $(K_1)K_2$ for the dependent product kind $(x{:}K_1)K_2$ in LF when x does not occur free in K_2. We have typed λ-terms of the form $[x{:}A]b$ (instead of the untyped $(x)b$).

[1] The filling-up rules are equivalent to the η-rules such as η for functions, surjective pairing for pairs.

K is called a Γ-*kind* if $\Gamma \vdash K$ **kind**. A is called a Γ-*type* if $\Gamma \vdash A : \mathbf{Type}$. K is called a *small* Γ-*kind* if $K \equiv (x_1{:}A_1)...(x_n{:}A_n)A_{n+1}$ such that $\Gamma, x_1{:}A_1, ..., x_{i-1}{:}A_{i-1} \vdash A_i : \mathbf{Type}$.

We take LF seriously as a *meta-language* to specify object languages, *i.e.*, a type theory in our case. A specification will consist of a set of declarations of constants and certain computation rules. In general, introducing a constant k of kind K and asserting a computation rule by '$k = k' : K$ for $k_i : K_i$ $(i = 1, ..., n)$' are to extend the LF system by the following rules, respectively:

$$\frac{\Gamma \ \mathbf{valid}}{\Gamma \vdash k : K} \qquad\qquad \frac{\Gamma \vdash k_i : K_i \ (i = 1, ..., n)}{\Gamma \vdash k = k' : K}$$

The specified type theory has five forms of judgements, which are

$$\Gamma \ \mathbf{valid} \quad \Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash A = B : \mathbf{Type} \quad \Gamma \vdash a : El(A) \quad \Gamma \vdash a = b : El(A)$$

where context Γ is of the form $x_1{:}El(A_1), ..., x_n{:}El(A_n)$. A judgement in the specified type theory is derivable if it is derivable in the LF extended by the constants and computation rules.

2.2 An impredicative universe of logical propositions

As an example of introducing types in LF, we introduce a type (impredicative universe) $Prop$ of logical propositions and the proof types of propositions, which, together with the Π-types to be introduced, give the higher-order logic embedded in the type theory. They are introduced by

$$\begin{array}{rcl} Prop & : & \mathbf{Type} \\ \mathbf{Prf} & : & (Prop)\mathbf{Type} \\ \forall & : & (A{:}\mathbf{Type})((A)Prop)Prop \\ \Lambda & : & (A{:}\mathbf{Type})(P{:}(A)Prop)((x{:}A)\mathbf{Prf}(P(x)))\mathbf{Prf}(\forall(A,P)) \\ \mathbf{E}_\forall & : & (A{:}\mathbf{Type})(P{:}(A)Prop)(R{:}(\mathbf{Prf}(\forall(A,P)))Prop) \\ & & ((g{:}(x{:}A)\mathbf{Prf}(P(x)))\mathbf{Prf}(R(\Lambda(A,P,g)))) \ (z{:}\mathbf{Prf}(\forall(A,P)))\mathbf{Prf}(R(z)) \end{array}$$

with computation rule

$$\mathbf{E}_\forall(A, P, R, f, \Lambda(A, P, g)) = f(g) : \mathbf{Prf}(R(\Lambda(A, P, g)))$$

We shall use usual notations later, *e.g.*, writing $\forall x{:}A.P(x)$ for $\forall(A, P)$. Note that the usual application operator,

$$\mathbf{App} \ : \ (A{:}\mathbf{Type})(P{:}(A)Prop)(\mathbf{Prf}(\forall(A,P)))(a{:}A)\mathbf{Prf}(P(a))$$

can be defined as

$$\mathbf{App}(A, P, F, a) =_{\mathrm{df}} \mathbf{E}_\forall(A, P, [G{:}\mathbf{Prf}(\forall(A,P))]P(a), [g{:}(x{:}A)\mathbf{Prf}(P(x))]g(a), F)$$

which satisfies the β-equality: $\mathbf{App}(A, P, \Lambda(A, P, g), a) = g(a) : \mathbf{Prf}(P(a))$.

2.3 Kind schemata in logical framework

In order to formulate schemata for inductive data types, we extend the logical framework by a notion of kind schema, which is very similar to Coquand and Mohring's notion of constructor in [CPM90], with the difference that we consider the schema in the logical framework, while they consider it based on the Π-types in the calculus of constructions with universes.

Definition 2.1 (kind schemata) *Let X be a fixed special symbol (a placeholder) and Γ be a valid context.*

- Φ *is a* strictly positive operator (in Γ)*, notation* $\mathrm{Pos}_\Gamma(\Phi)$*, if Φ is of the form* $(x_1{:}K_1)...(x_n{:}K_n)X$*, where $n \geq 0$ and K_i is a small* $(\Gamma, x_1{:}K_1, ..., x_{i-1}{:}K_{i-1})$*-kind.*

- Θ *is a* Γ-schema*, notation* $\mathrm{SCH}_\Gamma(\Theta)$*, if*

  1. $\Theta \equiv X$*, or*
  2. $\Theta \equiv (x{:}K)\Theta_0$*, where* K *is a small* Γ*-kind and* $\mathrm{SCH}_{\Gamma,x:K}(\Theta_0)$*, or*
  3. $\Theta \equiv (\Phi)\Theta_0$*, where* $\mathrm{POS}_\Gamma(\Phi)$ *and* $\mathrm{SCH}_\Gamma(\Theta_0)$.

Notations We shall use Φ, Φ', ... for strictly positive operators and Θ, Θ', ... for schemata. We shall also write $\bar{\Theta}$ for a sequence of schemata $\Theta_1, ..., \Theta_n$.

We write $\Phi(A)$ and $\Theta(A)$ for $[A/X]\Phi$ and $[A/X]\Theta$, respectively. For any small kind K, $K(A)$ is just K. Note that, for any (small) Γ-kind A, $\Phi(A)$ and $\Theta(A)$ are (small) Γ-kinds, if $\mathrm{POS}_\Gamma(\Phi)$ and $\mathrm{SCH}_\Gamma(\Theta)$.

Definition 2.2 (LF with schemata) *The* logical framework with schemata*,* LF_θ*, is the extension of* LF *by terms of the forms* $\mathcal{M}[\bar{\Theta}]$*,* $\iota_i[\bar{\Theta}]$ $(i = 1, ..., n)$*, and* $\mathbf{E}[\bar{\Theta}]$*, where* $\bar{\Theta} \equiv \Theta_1, ..., \Theta_n$ *is any finite sequence of schemata, and the following equality rule:*

$$\frac{\mathrm{SCH}_\Gamma(\bar{\Theta}, \bar{\Theta}') \quad \Gamma, A{:}\mathbf{Type} \vdash \bar{\Theta}(A) = \bar{\Theta}'(A) \quad \Gamma \vdash \kappa[\bar{\Theta}] : K}{\Gamma \vdash \kappa[\bar{\Theta}] = \kappa[\bar{\Theta}'] : K} \quad (\kappa \in \{\mathcal{M}, \iota_i, \mathbf{E}\})$$

where the second premise stands for n *premises* $\Gamma, A{:}\mathbf{Type} \vdash \Theta_i(A) = \Theta_i'(A)$*,* $(i = 1, ..., n)$.

3 Inductive Types

3.1 A schematic formulation of inductive types

The idea is that any finite sequence of schemata specifies a set of introduction rules and hence generates an inductive data type whose meaning is determined by the introduction rules. First, we introduce several notational definitions.

Definition 3.1 *Let* Φ *be a strictly positive operator.*

- *Define* $\Phi^\circ(A, C, z)$*, for* $A : \mathbf{Type}$*,* $C : (A)\mathbf{Type}$ *and* $z : \Phi(A)$*, by induction on the structure of* Φ *as follows:*

  1. *If* $\Phi \equiv X$*, then* $\Phi^\circ(A, C, z) =_{\mathrm{df}} C(z)$.
  2. *If* $\Phi \equiv (x{:}K)\Phi_0$*, then* $\Phi^\circ(A, C, z) =_{\mathrm{df}} (x{:}K)\Phi_0^\circ(A)(C, z(x))$.

- *Define* $\Phi^\natural(A) : (C{:}(A)\mathbf{Type})(f{:}(x{:}A)C(x))(z{:}\Phi(A))\Phi^\circ(A, C, z)$*, the functorial extension of* $\Phi(A)$*, by induction on the structure of* Φ *as follows:*

  1. *If* $\Phi \equiv X$*, then* $\Phi^\natural(A)(C, f) =_{\mathrm{df}} f$.
  2. *If* $\Phi \equiv (x{:}K)\Phi_0$*, then* $\Phi^\natural(A)(C, f, z) =_{\mathrm{df}} [x{:}K]\Phi_0^\natural(A)(C, f, z(x))$.

Definition 3.2 *Let* $\Theta \equiv (x_1{:}M_1)...(x_n{:}M_n)X$ *be a kind schema.*

- *The* arity *of* Θ*, notation* $\mathrm{ARI}(\Theta)$*, is the subsequence of* $\langle M_1, ..., M_n\rangle$ *consisting of the strictly positive operators. (This can be defined by induction on the structure of* Θ*.)*

- *Let* $\mathrm{ARI}(\Theta)$ *be* $\langle M_{i_1}, ..., M_{i_k}\rangle$*. Then, for* $A : \mathbf{Type}$*,* $C{:}(A)\mathbf{Type}$ *and* $z : \Theta(A)$*,*

$$\begin{aligned}\Theta^\circ(A, C, z) \quad =_{\mathrm{df}} \quad & (x_1{:}M_1(A))...(x_n{:}M_n(A)) \\ & (M_{i_1}^\circ(A, C, x_{i_1}))...(M_{i_k}^\circ(A, C, x_{i_k}))\ C(z(x_1, ..., x_n))\end{aligned}$$

Let $\bar{\Theta} \equiv \langle \Theta_1, ..., \Theta_n \rangle$ $(n \in \omega)$ be a sequence of Γ-schemata which have arities $\text{ARI}(\Theta_i) \equiv \langle \Phi_{i_1}, ..., \Phi_{i_k} \rangle$. Then, $\bar{\Theta}$ generates a Γ-type $\mathcal{M}[\bar{\Theta}]$ that is introduced by (with context Γ omitted)

$$\begin{array}{rcl} \mathcal{M}[\bar{\Theta}] & : & \mathbf{Type} \\ \iota_i[\bar{\Theta}] & : & \Theta_i(\mathcal{M}[\bar{\Theta}]) \quad (i = 1, ..., n) \\ \mathbf{E}[\bar{\Theta}] & : & (C{:}(\mathcal{M}[\bar{\Theta}])\mathbf{Type})\ (f_1{:}\Theta_1^\circ(\mathcal{M}[\bar{\Theta}], C, \iota_1[\bar{\Theta}])\ ...\ (f_n{:}\Theta_n^\circ(\mathcal{M}[\bar{\Theta}], C, \iota_n[\bar{\Theta}]) \\ & & (z{:}\mathcal{M}[\bar{\Theta}])C(z) \end{array}$$

with the computation rules $(i = 1, ..., n)$

$$\begin{array}{rl} & \mathbf{E}[\bar{\Theta}](C, \bar{f}, \iota_i(\bar{a})) \\ = & f_i(\bar{a}, \Phi_{i_1}^\natural(\mathcal{M}[\bar{\Theta}])(C, \mathbf{E}[\bar{\Theta}](C, \bar{f}), a_{i_1}), ..., \Phi_{i_k}^\natural(\mathcal{M}[\bar{\Theta}])(C, \mathbf{E}[\bar{\Theta}](C, \bar{f}), a_{i_k})) \\ : & C(\iota_i(\bar{a})) \end{array}$$

where $\bar{f}$ stands for $f_1, ..., f_n$ and $\bar{a}$ for $a_1, ..., a_n$.

3.2 Examples

The following are some examples covered by the schema for inductive types.

1. Empty type: $\emptyset =_{\text{df}} \mathcal{M}[]$.
2. Unit type: $\mathbf{1} =_{\text{df}} \mathcal{M}[X]$.
3. Natural numbers: $N =_{\text{df}} \mathcal{M}[X, (X)X]$.
4. Lists: $\mathit{List} =_{\text{df}} [A{:}\mathbf{Type}]\ \mathcal{M}[X, (A)(X)X]$.
5. Function space: $\rightarrow =_{\text{df}} [A{:}\mathbf{Type}][B{:}\mathbf{Type}]\ \mathcal{M}[((A)B)X]$.
6. Dependent product: $\Pi =_{\text{df}} [A{:}\mathbf{Type}][B{:}(A)\mathbf{Type}]\ \mathcal{M}[((x{:}A)B(x))X]$.
7. Product: $\times =_{\text{df}} [A{:}\mathbf{Type}][B{:}\mathbf{Type}]\ \mathcal{M}[(A)(B)X]$.
8. Strong sum: $\Sigma =_{\text{df}} [A{:}\mathbf{Type}][B{:}(A)\mathbf{Type}]\ \mathcal{M}[(x{:}A)(B(x))X]$.
9. Disjoint sum: $+ =_{\text{df}} [A{:}\mathbf{Type}][B{:}\mathbf{Type}]\ \mathcal{M}[(A)X, (B)X]$.
10. Well-ordering: $W =_{\text{df}} [A{:}\mathbf{Type}][B{:}(A)\mathbf{Type}]\ \mathcal{M}[(x{:}A)((B(x))X)X]$.

Other examples like binary trees, ordinals, *etc.* can be similarly defined. Note that, these definitions give desirable rules for the corresponding types. For example, one may easily check that the above type constructors have exactly the same rules as those given in [NPS90].

3.3 Intensionality and filling-up rules

The computational equality between objects of the inductive types introduced by the schemata are intensional, which in our view, captures the notion of computation in a satisfactory way. For example, the η-rule does not hold in general for the functions of type $\Pi(A, B)$, although they are true for closed functions. It is worth remarking that, although they do not hold computationally, the filling-up equality rules are in fact valid *logically*.

Proposition 3.3 (logical validity of filling-up rules) *Let $C : (\mathcal{M}[\bar{\Theta}])\mathbf{Type}$ and $f : (z{:}\mathcal{M}[\bar{\Theta}])C(z)$, where $\text{SCH}(\bar{\Theta})$. Then, the following proposition is provable (i.e., its proof type is inhabited):*

$$\forall u{:}\mathcal{M}[\bar{\Theta}].\ (f(u) =_{C(u)} \mathbf{E}[\bar{\Theta}](C, f \circ \bar{\iota}, u))$$

where $=_{C(u)}$ *is the Leibniz's equality*[2] *over* $C(u)$*,* $f \circ \bar{\iota}$ *stands for* $f \circ \iota_1, ..., f \circ \iota_n$ *and* $(f \circ \iota_i)(a_1, ..., a_n, y_1, ..., y_k) =_{\mathrm{df}} f(\iota_i(a_1, ..., a_n))$.

Since the usual 'η-rules' (η for Π, surjective pairing for Σ, *etc.*) are equivalent to the filling-up rules, the above proposition shows that, for the inductive data types in general, the η-rules hold logically (for the Leibniz's equality). Note that the η-rules express that every object of an inductive type is equal to a canonical object and the filling-up rules express that the elimination operator covers all of the use of the inductive type. From this point of view, the above proposition may be regarded as an internal (or logical) justification of the adequacy of the formulation of inductive types, in particular, the intrinsic harmony between the introduction and elimination rules (*c.f.*, [Dum91]).

The above fact is also a benefit of using a logical framework to define an intensional type theory with a general schema of inductive types. It seems impossible to prove the η-rule for the Leibniz's equality if we do not use a meta-language to formulate Π-types (in the case of the direct formulation of **ECC**, for instance), unless one introduces η as a computational rule for the Π-types, which would destroy the intensionality of the theory.

4 Predicative Universes and Reflection Principle

We introduce predicative universes

$$Type_i : \mathbf{Type} \quad (i \in \omega)$$

with lifting operators

$$\mathbf{T}_i : (Type_i)\mathbf{Type} \qquad \mathbf{t}_{i+1} : (Type_i)Type_{i+1} \qquad \mathbf{t}_0 : (Prop)Type_0$$

The reflection rules about the predicative universes and the impredicative universe and propositions are the following:

$$type_i : Type_{i+1} \qquad prop : Type_0$$

$$\mathbf{T}_{i+1}(type_i) = Type_i : \mathbf{Type} \qquad \mathbf{T}_0(prop) = Prop : \mathbf{Type}$$

$$\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a) : \mathbf{Type} \qquad \mathbf{T}_0(\mathbf{t}_0(P)) = \mathbf{Prf}(P) : \mathbf{Type}$$

where $a : Type_i$ and $P : Prop$.

Now, we consider the reflection of the inductive types introduced by the schemata.

Definition 4.1 *Let K, Φ and Θ be a small Γ-kind, a strictly positive operator in Γ and a Γ-schema, respectively. The sets* $\textsc{Types}_\Gamma(K)$*,* $\textsc{Types}_\Gamma(\Phi)$ *and* $\textsc{Types}_\Gamma(\Theta)$ *are defined as follows:*

$$\textsc{Types}_\Gamma(K) =_{\mathrm{df}} \begin{cases} \{(\Gamma, K)\} & \textit{if } K \textit{ is a } \Gamma\textit{-type} \\ \textsc{Types}_\Gamma(K_1) \cup \textsc{Types}_{\Gamma, x:K_1}(K_2) & \textit{if } K \equiv (x{:}K_1)K_2 \end{cases}$$

$$\textsc{Types}_\Gamma(\Phi) =_{\mathrm{df}} \begin{cases} \emptyset & \textit{if } \Phi \equiv X \\ \textsc{Types}_\Gamma(K_1) \cup \textsc{Types}_{\Gamma, x:K_1}(\Phi_0) & \textit{if } \Phi \equiv (x{:}K_1)\Phi_0 \end{cases}$$

$$\textsc{Types}_\Gamma(\Theta) =_{\mathrm{df}} \begin{cases} \emptyset & \textit{if } \Theta \equiv X \\ \textsc{Types}_\Gamma(K_1) \cup \textsc{Types}_{\Gamma, x:K_1}(\Theta_0) & \textit{if } \Theta \equiv (x{:}K_1)\Theta_0 \\ \textsc{Types}_\Gamma(\Phi_1) \cup \textsc{Types}_\Gamma(\Theta_0) & \textit{if } \Theta \equiv (\Phi_1)\Theta_0 \end{cases}$$

For $\bar{\Theta} \equiv \Theta_1, ..., \Theta_n$*,* $\textsc{Types}_\Gamma(\bar{\Theta}) =_{\mathrm{df}} \bigcup_{1 \le i \le n} \textsc{Types}_\Gamma(\Theta_i)$.

[2] The Leibniz's equality is defined as: for any type A and any objects a and b of type A, $(a =_A b) =_{\mathrm{df}} \forall P{:}A \to Prop.\ \mathbf{app}(A, Prop, P, a) \supset \mathbf{app}(A, Prop, P, b)$, where **app** is the application operator for the function type and $\supset$ is the logical implication operator.

Let $\bar{\Theta}$ be a finite sequence of Γ-schemata. Then, if, for any $(\Gamma', A) \in \textsc{Types}_\Gamma(\bar{\Theta})$, there exists an a such that

$$\Gamma' \vdash a : Type_i \quad \text{and} \quad \Gamma' \vdash \mathbf{T}_i(a) = A : \mathbf{Type}$$

we have the following reflection rules for the inductive type $\mathcal{M}[\bar{\Theta}]$ (we omit context Γ):[3]

$$\Gamma \vdash \mu_i[\bar{\Theta}] : Type_i$$

$$\Gamma \vdash \mathbf{T}_i(\mu_i[\bar{\Theta}]) = \mathcal{M}[\bar{\Theta}] : \mathbf{Type}$$

$$\Gamma \vdash \mathbf{T}_{i+1}(\mathbf{t}_{i+1}(\mu_i[\bar{\Theta}])) = \mathbf{T}_i(\mu_i[\bar{\Theta}]) : \mathbf{Type}$$

For example, the type of natural numbers has a name in any predicative universe $Type_i$. For the Π-types, $\Pi(A, B)$ has a name in $Type_i$ if and only if A has a name in $Type_i$ and $B(x)$ has a name in $Type_i$, assuming $x{:}A$.

Remark Using a logical framework with schemata, we have introduced inductive data types independently with the existence of the predicative type universes. In other words, the inductive types exist independently with predicative universes and their names are introduced when universes are introduced. Note that the type universes in our theory are *not* inductively defined since there is no elimination rules to impose induction principle over them. In fact, being aware of the potential infinity of conceptions in type formation, it does not seem to be reasonable to consider such universes as closed for finitely many type constructors.

Using predicative universes, one can define families of types. For example, supposing type A has a name in $Type_0$, we can define a function $Listn$ of type $N \to Type_0$ such that, for natural number $n : N$, $Listn(n)$ is the (name of the) inductive type of lists of objects in A of length n. $Listn$ can be defined by induction over N as $Listn(0) = \mu_0[X]$ and $Listn(n+1) = \mu_0[(A)(\mathbf{T}_0(Listn(n)))X]$.

5 Summary and Discussions

We have presented a theory of dependent types (call it UTT) which consists of the impredicative universe (see section 2.2), a class of inductive types covered by a general form of schemata (see section 3.1), and predicative universes (see section 4). It may be seen as an extension of **ECC** by a large class of inductive data types, and we claim that, as **ECC**, UTT has nice meta-theoretic properties. The realizability model in [Luo91a] can be extended to UTT (*c.f.*, [Ore90, CPM90]) to show its model-theoretic consistency. We also conjecture that UTT has the Church-Rosser and strong normalization properties (subject to the obvious notion of reduction) and is decidable. The strong normalization theorem may be proved by extending the method of quasi-normalization used to prove strong normalization of **ECC** in [Luo90a]. Also, the notion of head-normal form as investigated in [Coq91] can be used to study the meta-theory and type-checking algorithm for UTT. It is expected that the intensionality of the system should make the study of its meta-theory easier than the systems with filling-up rules such as that considered in [GL91], and it seems that the separation between the meta-level (logical framework) and the type theory is beneficial as well.

Now, we discuss several issues about the theory that we think interesting for further research.

5.1 Compositional understanding

A verificationistic meaning theory may be given for the type theory UTT, using a proof-theoretic justification method based on the notion of computation and a notion of canonical object, in a similar way as explained by Dummett [Dum75, Dum91], Prawitz [Pra74], and particularly, Martin-Löf [ML84]. It would be interesting to see whether the language of the type theory allows a *compositional* understanding in the sense of Dummett [Dum91]. The following notion and conjecture of logical conservativity seems to be interesting to consider in this aspect.

[3]We may also include rules of name uniqueness for inductive types: $\Gamma \vdash \mathbf{t}_{i+1}(\mu_i[\bar{\Theta}]) = \mu_{i+1}[\bar{\Theta}] : Type_{i+1}$.

Definition 5.1 (logical conservativity) *Let T and T' be type systems specified in* LF*, both of which contain the impredicative universe specified in section 2.2, such that T is a subsystem of T'. Then, T'* is logically conservative over *T if and only if, for any proposition P (of type $Prop$) in T, if P is provable in T', then P is provable in T.*

Conjecture 5.2 (logical conservativity) *Let UTT_0 be the type theory UTT without predicative universes, UTT_{i+1} be the extension of UTT_i by the ith predicative universe $Type_i$ ($i \in \omega$), and T be any subsystem of UTT_0 containing the impredicative universe specified in section 2.2 and a finite set of inductive types specified by kind schemata. Then, we conjecture that the following hold:*

1. *LC_1: The extension of T by any inductive type $\mathcal{M}[\bar{\Theta}]$ is logically conservative over T.*

2. *LC_2: UTT_{i+1} is logically conservative over UTT_i for $i \in \omega$.*

The above conjecture, if true, allows us to understand the language of UTT in a compositional or hierarchical way and will have interesting impact on learning and using the theory. It may be the case that the conjecture is too strong and that one needs to understand some types simultaneously. Either a proof of the conjecture or a counter-example against it (which might then lead to a more proper way in understanding) will enhance the understanding of the type theory.

5.2 Inductive relations

Note that in our formulation we have not included more general schemata to cover general inductive relations or inductive families of types (*c.f.*, [CPM90, Dyb91]). For example, we do not seem to be able to define Martin-Löf's weak equality types $Eq(A, a, b)$ in UTT. Although it is possible to extend the notion of kind schemata to do so, there seem to be good reasons in favor of the more moderate approach. First, since we distinguish the notions of logical propositions and data types, logical inductive relations can be defined by impredicative definitions. For instance, a logical equality should be of type $A \to A \to Prop$ in our theory and can be defined impredicatively (*e.g.*, Leibniz's equality). Second, many families of types can be defined using predicative universes (or parametric definitions in the framework). *Listn* defined at the end of section 4 is an example of inductive family of types. Having said these, it is ceratinly interesting to investigate more general forms of schemata for more sophisticated inductive types.

Another consideration to exclude families of types like $Eq(A)$ concerns with the desire to gain a compositional understanding of the language as explained above. Including inductive families of types like $Eq(A)$ would make a hierarchical understanding of such a theory much more difficult. This can be seen from Smith's proof that adding a universe to Martin-Löf's type theory without universes (but with the equality types) is not conservative [Smi88]. If we included Martin-Löf's weak equality types as inductive types in UTT, the conjecture LC_2 would fail to hold.

5.3 Subtyping

An interesting application of the general notion of schemata is that it provides a general guideline to introduce subtyping relations between inductive types. For example, the subtyping relation between Π and Σ-types in **ECC**, induced by the (Russell-style) universe inclusions, is a special case of a general notion of subtyping between inductive types. To see this, define a partial order $<$ between schemata and the subtyping relation $\preceq$ (a partial order subject to the computational equality) by simultaneous induction as follows: $\Theta < \Theta'$ if and only if

1. $\Theta \equiv \Theta' \equiv X$; or

2. $\Theta \equiv (x{:}K)\Theta_1$ and $\Theta' \equiv (x{:}K')\Theta'_1$ with $K \equiv (x_1{:}A_1)...(x_m{:}A_m)A$ and $K' \equiv (x_1{:}A'_1)...(x_m{:}A'_m)A'$, and $A_i = A'_i$ for $i = 1, ..., m$, $A \preceq A'$ and $\Theta_1 < \Theta'_1$; or

3. $\Theta \equiv (\Phi)\Theta_1$, $\Theta' \equiv (\Phi')\Theta'_1$, and $\Phi(A) = \Phi'(A)$ for A:**Type**, and $\Theta_1 < \Theta'_1$.

And, $\mathcal{M}[\bar{\Theta}] \preceq \mathcal{M}[\bar{\Theta}']$ if and only if $\Theta_i < \Theta_i'$ for $i = 1, ..., n$. It is easy to check that this gives the desirable subtyping for the type constructors such as Π, Σ, $+$ and W.

5.4 Proofs and implementation

Regarding the propositions of type $Prop$ in UTT, we remark that there is a stronger elimination operator for the proof types which might be introduced as:

$$\begin{array}{rcl}\mathbf{E}'_\forall & : & (A{:}\mathbf{Type})(P{:}(A)Prop)(C{:}(\mathbf{Prf}(\forall(A,P)))\mathbf{Type}) \\ & & ((g{:}(x{:}A)\mathbf{Prf}(P(x)))C(\Lambda(A,P,g)))\ (z{:}\mathbf{Prf}(\forall(A,P)))C(z)\end{array}$$

$\mathbf{E}'_\forall$ is different from $\mathbf{E}_\forall$ in that, with $\mathbf{E}'_\forall$, one can define functions from a proof type to any type, instead of just to proof types. It may be interesting to investigate this to see whether it can provide a stronger version of the theory to manipulate proofs.

Defining UTT by a logical framework also raises an interesting issue in implementation of proof development systems like Lego [Pol89, LPT89]. Although the notion of computational equality (between objects of types) is intensional, it seems nice to have a weakly extensional meta-level definitional mechanism in the sense that η-rule holds for the meta-level functional operations. Note that, although one might think that the Π-types are the internal version of the dependent product kinds in the logical framework, there is no one-one correspondence between the objects of type $\Pi(A,B)$ and the objects of kind $(x{:}A)B(x)$, as the former is the type of *intensional* functions while the latter is the kind of *extensional* operations. We also remark that the separation of meta-reasoning (in LF in our case) from reasoning in the type theory is important.

Acknowledgement I would like to thank Per Martin-Löf, Thierry Coquand and Peter Dybjer for interesting conversations on related topics and Healfdene Goguen for many helpful discussions.

References

[Bac88] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In A. Avron et al, editor, *Workshop on General Logic.* LFCS Report Series, ECS-LFCS-88-52, Dept. of Computer Science, University of Edinburgh, 1988.

[BM91] R. Burstall and J. McKinna. Deliverables: an approach to program development in the calculus of constructions. LFCS report ECS-LFCS-91-133, Dept of Computer Science, University of Edinburgh, 1991.

[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.

[Coq91] Th. Coquand. An algorithm for testing conversion in Type Theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991.

[CPM90] Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[Dum75] M. Dummett. The philosophical basis of intuitionistic logic. In H. Rose and J. Shepherdson, editors, *Proc. of the Logic Colloquium, 1973.* North Holland, 1975. Reprinted in P. Benacerraf and H. Putnam (eds.), Philosophy of Mathematics: selected readings, Campbridge University Press.

[Dum91] M. Dummett. *The Logical Basis of Metaphysics.* Duckworth, 1991.

[Dyb89] P. Dybjer. An inversion principle for Martin-Löf's type theory. In P. Dybjer et al, editor, *Workshop on Programming Logic*. Programming Methodology Group, Report 54, 1989.

[Dyb91] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[GL91] H. Goguen and Z. Luo. Inductive data types: Well-ordering types revisited. submitted manuscript, 1991.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Proc. 2nd Ann. Symp. on Logic in Computer Science*, 1987.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. *How to Use LEGO: a preliminary user's manual.* LFCS Technical Notes LFCS-TN-27, Dept. of Computer Science, Edinburgh University, 1989.

[Luo89] Z. 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.

[Luo90a] Z. 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.

[Luo90b] Z. Luo. A problem of adequacy: conservativity of calculus of constructions over higher-order logic. Technical report, LFCS report series ECS-LFCS-90-121, Department of Computer Science, University of Edinburgh, 1990.

[Luo91a] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1):107–137, 1991.

[Luo91b] Z. Luo. Program specification and data refinement in type theory. *Proc. of the Fourth Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT)*, 1991. Also as LFCS report ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University.

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

[ML84] P. 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.

[Ore90] C.-E. Ore. The Extended Calculus of Constructions (**ECC**) with inductive types. To appear in Information and Computation, 1990.

[Pol89] R. Pollack. The theory of LEGO. manuscript, 1989.

[Pol90] R. Pollack. The Tarski fixpoint theorem. communication on TYPES e-mail network, 1990.

[Pra74] D. Prawitz. On the idea of a general proof theory. *Synthese*, 27, 1974.

[Smi88] J. Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3), 1988.

A The Inference Rules of LF

contexts and assumptions

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

general equality rules

$$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash K = K} \qquad \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \qquad \frac{\Gamma \vdash K = K' \quad \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash k = k : K} \qquad \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k : K} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' : K}$$

equality typing

$$\frac{\Gamma \vdash k : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k : K'} \qquad \frac{\Gamma \vdash k = k' : K \quad \Gamma \vdash K = K'}{\Gamma \vdash k = k' : K'}$$

substitution rules

$$\frac{\Gamma, x{:}K, \Gamma'\ \textbf{valid} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma'\ \textbf{valid}}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash K'\ \textbf{kind} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K'\ \textbf{kind}} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash K'\ \textbf{kind} \quad \Gamma \vdash k = k' : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k'/x]K'}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' : [k/x]K'} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k = k' : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' = [k'/x]k' : [k/x]K'}$$

$$\frac{\Gamma, x{:}K, \Gamma' \vdash K' = K'' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k/x]K''} \qquad \frac{\Gamma, x{:}K, \Gamma' \vdash k' = k'' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' = [k/x]k'' : [k/x]K'}$$

dependent product kinds

$$\frac{\Gamma \vdash K\ \textbf{kind} \quad \Gamma, x{:}K \vdash K'\ \textbf{kind}}{\Gamma \vdash (x{:}K)K'\ \textbf{kind}} \qquad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash K_1' = K_2'}{\Gamma \vdash (x{:}K_1)K_1' = (x{:}K_2)K_2'}$$

$$\frac{\Gamma, x{:}K \vdash k : K'}{\Gamma \vdash [x{:}K]k : (x{:}K)K'} \qquad \frac{\Gamma \vdash K_1 = K_2 \quad \Gamma, x{:}K_1 \vdash k_1 = k_2 : K}{\Gamma \vdash [x{:}K_1]k_1 = [x{:}K_2]k_2 : (x{:}K_1)K}$$

$$\frac{\Gamma \vdash f : (x{:}K)K' \quad \Gamma \vdash k : K}{\Gamma \vdash f(k) : [k/x]K'} \qquad \frac{\Gamma \vdash f = f' : (x{:}K)K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$$

$$\frac{\Gamma, x{:}K \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma \vdash ([x{:}K]k')(k) = [k/x]k' : [k/x]K'} \qquad \frac{\Gamma \vdash f : (x{:}K)K' \quad x \notin FV(f)}{\Gamma \vdash [x{:}K]f(x) = f : (x{:}K)K'}$$

Type rules

$$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash \textbf{Type kind}} \qquad \frac{\Gamma \vdash A : \textbf{Type}}{\Gamma \vdash El(A)\ \textbf{kind}}$$