A modular construction of type theories

Frédéric Blanqui, Gilles Dowek, Emilie Grienenberger, Gabriel Hondet, François Thiré

Logical Methods in Computer Science
Volume 19, Issue 1, 2023, pp. 12:1–12:28
https://lmcs.episciences.org/

Submitted Nov. 02, 2021
Published Feb. 14, 2023

A MODULAR CONSTRUCTION OF TYPE THEORIES

FRÉDÉRIC BLANQUI [a], GILLES DOWEK [a], EMILIE GRIENENBERGER [a], GABRIEL HONDET [a], AND FRANÇOIS THIRÉ [b]

[a] Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France
e-mail address: {frederic.blanqui,gilles.dowek,gabriel.hondet}@inria.fr
e-mail address: emilie.grienenberger@ens-paris-saclay.fr

[b] Nomadic Labs, France
e-mail address: francois.thire@nomadic-labs.com

Abstract. The λΠ-calculus modulo theory is a logical framework in which many type systems can be expressed as theories. We present such a theory, the theory $\mathcal{U}$, where proofs of several logical systems can be expressed. Moreover, we identify a sub-theory of $\mathcal{U}$ corresponding to each of these systems, and prove that, when a proof in $\mathcal{U}$ uses only symbols of a sub-theory, then it is a proof in that sub-theory.

1. Introduction

The λΠ-calculus modulo theory (λΠ/≡) [CD07], that is the basis of the language Dedukti [ABC+16, HB20], is a logical framework, that is, a framework to define theories. It generalizes some previously proposed frameworks: Predicate logic [HA28], λ-Prolog [NM88], Isabelle [Pau93], the Edinburgh logical framework [HHP93], also called the λΠ-calculus, Deduction modulo theory [DHK03, DW03], Pure type systems [Ber88, Ter89], and Ecumenical logic [Pra15, Dow15, PR17, Gri19]. It is thus an extension of Predicate logic that provides the possibility for all symbols to bind variables, a syntax for proof-terms, a notion of computation, a notion of proof reduction for axiomatic theories, and the possibility to express both constructive and classical proofs.

λΠ/≡ enables to express all theories that can be expressed in Predicate logic, such as geometry, arithmetic, and set theory, but also Simple type theory [Chu40] and the Calculus of constructions [CH88], that are less easy to define in Predicate logic.

We present a theory in λΠ/≡, the theory $\mathcal{U}$, where all proofs of Minimal, Constructive, and Ecumenical predicate logic; Minimal, Constructive, and Ecumenical simple type theory; Simple type theory with predicate subtyping, prenex predicative polymorphism, or both; the Calculus of constructions, and the Calculus of constructions with prenex predicative polymorphism can be expressed. This theory is therefore a candidate for a universal theory, where proofs developed in implementations of Classical predicate logic (such as automated theorem proving systems, SMT solvers, etc.), Classical simple type theory (such as HOL 4, Isabelle/HOL, HOL Light, etc.), the Calculus of constructions (such as Coq, Matita, Lean,

Key words and phrases: logical framework, λΠ-calculus modulo rewriting, Dedukti, logic, type theory.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-19(1:12)2023

etc.), and Simple type theory with predicate subtyping and prenex polymorphism (such as PVS), can be expressed.

Moreover, the proofs of the theory $\mathcal{U}$ can be classified as proofs in Minimal predicate logic, Constructive predicate logic, etc. just by identifying the axioms they use, akin to proofs in geometry that can be classified as proofs in Euclidean, hyperbolic, elliptic, neutral, etc. geometries. More precisely, we identify sub-theories of the theory $\mathcal{U}$ that correspond to each of these theories, and we prove that when a proof in $\mathcal{U}$ uses only symbols of a sub-theory, then it is a proof in that sub-theory.

In Section 2, we recall the definition of $\lambda\Pi/\equiv$ and of a theory. In Section 3, we introduce the theory $\mathcal{U}$ step by step without any (other than informal) reference to a known system. In Section 4, we provide a general theorem on sub-theories in $\lambda\Pi/\equiv$, and prove that every fragment of $\mathcal{U}$, including $\mathcal{U}$ itself, is indeed a theory, that is, it is defined by a confluent and type-preserving rewriting system. Finally, in Section 5, we detail the sub-theories of $\mathcal{U}$ that correspond to the above mentioned systems.

This paper is an extended version of [BDG+21]. It provides more details on the encoding of various features (Section 3), the proof of the Fragment Theorem (Section 4), and the examples of sub-theories (Section 5).

2. The λΠ-calculus modulo theory

$\lambda\Pi/\equiv$ is an extension of the Edinburgh logical framework [HHP93] with a primitive notion of computation defined with rewriting rules [DJ90, TeR03].

However, for defining terms, we use Barendregt's syntax for Pure Type Systems [Bar92]:

$$t, u = c \mid x \mid \texttt{TYPE} \mid \texttt{KIND} \mid \Pi x : t, u \mid \lambda x : t, u \mid t\ u$$

where c belongs to a finite or infinite set of constants $\mathcal{C}$ and x to an infinite set $\mathcal{V}$ of variables. The terms `TYPE` and `KIND` are called sorts. The term $\Pi x : t, u$ is called a product. It is dependent if the variable x occurs free in u. Otherwise, it is simply written $t \to u$. Terms are also often written A, B, etc. The set of constants of a term t is written $\mathit{const}(t)$.

A rewriting rule is a pair of terms $\ell \hookrightarrow r$, such that $\ell = c\ \ell_1\ \ldots\ \ell_n$, where c is a constant. If $\mathcal{R}$ is a set of rewriting rules, we write $\hookrightarrow_{\mathcal{R}}$ for the smallest relation closed by term constructors and substitution containing $\mathcal{R}$, $\hookrightarrow_\beta$ for the usual β-reduction, $\hookrightarrow_{\beta\mathcal{R}}$ for $\hookrightarrow_\beta \cup \hookrightarrow_{\mathcal{R}}$, and $\equiv_{\beta\mathcal{R}}$ for the smallest equivalence relation containing $\hookrightarrow_{\beta\mathcal{R}}$.

A relation $\hookrightarrow$ is confluent on a set of terms if, for all terms t, u, v in this set such that $t \hookrightarrow^* u$ and $t \hookrightarrow^* v$, there exists a term w in this set such that $u \hookrightarrow^* w$ and $v \hookrightarrow^* w$. Confluence implies that every term has at most one irreducible form.

The typing rules of $\lambda\Pi/\equiv$ are given in Figure 1. The difference with the rules of the Edinburgh logical framework is that, in the rule (conv), types are identified modulo $\equiv_{\beta\mathcal{R}}$ instead of just $\equiv_\beta$. In a typing judgement $\Gamma \vdash_{\Sigma,\mathcal{R}} t : A$, the term t is given the type A with respect to three parameters: a signature Σ that assigns a type to the constants of t, a context Γ that assigns a type to the free variables of t, and a set of rewriting rules $\mathcal{R}$. A context Γ is a list of declarations $x_1 : B_1, \ldots, x_m : B_m$ formed with a variable and a term. A signature Σ is a list of declarations $c_1 : A_1, \ldots, c_n : A_n$ formed with a constant and a closed term, that is a term with no free variables. This is why the rule (const) requires no context for typing A. We write $|\Sigma|$ for the set $\{c_1, \ldots, c_n\}$, and $\Lambda(\Sigma)$ for the set of terms t such that $\mathit{const}(t) \subseteq |\Sigma|$. We say that a rewriting rule $\ell \hookrightarrow r$ is in $\Lambda(\Sigma)$ if ℓ and r are, and a context

$$\frac{}{\vdash_{\Sigma,\mathcal{R}} [\,] \text{ well-formed}}\ (\text{empty})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} A : s}{\vdash_{\Sigma,\mathcal{R}} \Gamma, x : A \text{ well-formed}}\ (\text{decl})$$

$$\frac{\vdash_{\Sigma,\mathcal{R}} \Gamma \text{ well-formed}}{\Gamma \vdash_{\Sigma,\mathcal{R}} \texttt{TYPE} : \texttt{KIND}}\ (\text{sort})$$

$$\frac{\vdash_{\Sigma,\mathcal{R}} \Gamma \text{ well-formed} \quad \vdash_{\Sigma,\mathcal{R}} A : s}{\Gamma \vdash_{\Sigma,\mathcal{R}} c : A}\ (\text{const})\ c : A \in \Sigma$$

$$\frac{\vdash_{\Sigma,\mathcal{R}} \Gamma \text{ well-formed}}{\Gamma \vdash_{\Sigma,\mathcal{R}} x : A}\ (\text{var})\ x : A \in \Gamma$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} A : \texttt{TYPE} \quad \Gamma, x : A \vdash_{\Sigma,\mathcal{R}} B : s}{\Gamma \vdash_{\Sigma,\mathcal{R}} \Pi x : A, B : s}\ (\text{prod})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} A : \texttt{TYPE} \quad \Gamma, x : A \vdash_{\Sigma,\mathcal{R}} B : s \quad \Gamma, x : A \vdash_{\Sigma,\mathcal{R}} t : B}{\Gamma \vdash_{\Sigma,\mathcal{R}} \lambda x : A, t : \Pi x : A, B}\ (\text{abs})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} t : \Pi x : A, B \quad \Gamma \vdash_{\Sigma,\mathcal{R}} u : A}{\Gamma \vdash_{\Sigma,\mathcal{R}} t\ u : (u/x)B}\ (\text{app})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathcal{R}} t : A \quad \Gamma \vdash_{\Sigma,\mathcal{R}} B : s}{\Gamma \vdash_{\Sigma,\mathcal{R}} t : B}\ (\text{conv})\ A \equiv_{\beta\mathcal{R}} B$$

Figure 1: Typing rules of $\lambda\Pi/\equiv$ with signature Σ and rewriting rules $\mathcal{R}$

$x_1 : B_1, \ldots, x_m : B_m$ is in $\Lambda(\Sigma)$ if $B_1, \ldots, B_m$ are. It is often convenient to group constant declarations and rules into small clusters, called "axioms".

A relation $\hookrightarrow$ preserves typing in $(\Sigma, \mathcal{R})$ if, for all contexts Γ and terms t, u and A of $\Lambda(\Sigma)$, if $\Gamma \vdash_{\Sigma,\mathcal{R}} t : A$ and $t \hookrightarrow u$, then $\Gamma \vdash_{\Sigma,\mathcal{R}} u : A$. The relation $\hookrightarrow_\beta$ preserves typing as soon as $\hookrightarrow_{\beta\mathcal{R}}$ is confluent (see for instance [Bla01]) for, in this case, the product is injective modulo $\equiv_{\beta\mathcal{R}}$: $\Pi x : A, B \equiv_{\beta\mathcal{R}} \Pi x : A', B'$ if and only if $A \equiv_{\beta\mathcal{R}} A'$ and $B \equiv_{\beta\mathcal{R}} B'$. The relation $\hookrightarrow_{\mathcal{R}}$ preserves typing if every rewriting rule $\ell \hookrightarrow r$ preserves typing, that is: for all contexts Γ, substitutions θ and terms A of $\Lambda(\Sigma)$, if $\Gamma \vdash_{\Sigma,\mathcal{R}} \theta l : A$ then $\Gamma \vdash_{\Sigma,\mathcal{R}} \theta r : A$.

Although typing is defined with arbitrary signatures Σ and sets of rewriting rules $\mathcal{R}$, we are only interested in sets $\mathcal{R}$ verifying some confluence and type-preservation properties.

Definition 2.1 (System, theory)**.** A system is a pair $(\Sigma, \mathcal{R})$ such that each rule of $\mathcal{R}$ is in $\Lambda(\Sigma)$. It is a theory if $\hookrightarrow_{\beta\mathcal{R}}$ is confluent on $\Lambda(\Sigma)$, and every rule of $\mathcal{R}$ preserves typing in $(\Sigma, \mathcal{R})$.

Therefore, in a theory, $\hookrightarrow_{\beta\mathcal{R}}$ preserves typing since $\hookrightarrow_\beta$ preserves typing (for $\hookrightarrow_{\beta\mathcal{R}}$ is confluent) and $\hookrightarrow_{\mathcal{R}}$ preserves typing (for every rule preserves typing). We recall two other basic properties of $\lambda\Pi/\equiv$ we will use in Theorem 4.5:

Lemma 2.2. *If $\Gamma \vdash_{\Sigma,\mathcal{R}} t : A$, then either $A = \texttt{KIND}$ or $\Gamma \vdash_{\Sigma,\mathcal{R}} A : s$ for some sort s.*

If $\Gamma \vdash_{\Sigma,\mathcal{R}} \Pi x : A, B : s$, then $\Gamma \vdash_{\Sigma,\mathcal{R}} A : \texttt{TYPE}$.

3. The theory $\mathcal{U}$

Let us now present the system $\mathcal{U}$ which is formed with axioms expressed in $\lambda\Pi/\equiv$. We will prove in Theorem 4.7 that this system is indeed a theory.

3.1. **Object-terms.** The notions of term, proposition, and proof are not primitive in $\lambda\Pi/\equiv$. The first axioms of the theory $\mathcal{U}$ introduce these notions. We first define a notion analogous to the Predicate logic notion of term, to express the objects the theory speaks about, such as the natural numbers. As all expressions in $\lambda\Pi/\equiv$ are called "terms", we shall call these expressions "object-terms", to distinguish them from the other terms.

To build the notion of object-term in $\lambda\Pi/\equiv$ we declare a constant I of type `TYPE`

$$I : \texttt{TYPE} \qquad (I\text{-decl})$$

and constants of type $I \to ... \to I \to I$ for the function symbols, for instance a constant 0 of type I and a constant *succ* of type $I \to I$. The object-terms, for instance $(succ\ (succ\ 0))$ and $(succ\ x)$, are then just $\lambda\Pi/\equiv$ terms of type I and, in an object-term, the variables are $\lambda\Pi/\equiv$ variables of type I. If we wanted to have object-terms of several sorts, like in Many-sorted predicate logic, we could just declare several constants I_1, I_2, ..., I_n of type `TYPE`.

3.2. **Propositions.** Just like $\lambda\Pi/\equiv$ does not contain a primitive notion of object-term, it does not contain a primitive notion of proposition, but tools to define this notion. To do so, in the theory $\mathcal{U}$, we declare a constant *Prop* of type `TYPE`

$$Prop : \texttt{TYPE} \qquad (Prop\text{-decl})$$

and predicate symbols, that is constants of type $I \to ... \to I \to Prop$, for instance a constant *positive* of type $I \to Prop$. Propositions are then $\lambda\Pi/\equiv$ terms, such as $(positive\ (succ\ (succ\ 0)))$, of type *Prop*.

3.3. **Implication.** In the theory $\mathcal{U}$, we then declare a constant for implication

$$\Rightarrow : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\Rightarrow\text{-decl})$$

and we can then construct the term $(positive\ (succ\ (succ\ 0))) \Rightarrow (positive\ (succ\ (succ\ 0)))$ of type *Prop*.

3.4. **Proofs.** Predicate logic defines a language for terms and propositions, but proofs have to be defined in a second step, for instance as derivations in natural deduction, sequent calculus, etc. These derivations, like object-terms and propositions, are trees. Therefore, they can be represented as $\lambda\Pi/\equiv$ terms.

Using the Brouwer-Heyting-Kolmogorov interpretation, a proof of the proposition $A \Rightarrow B$ should be a $\lambda\Pi/\equiv$ term expressing a function mapping proofs of A to proofs of B. Then, using the Curry-de Bruijn-Howard correspondence, the type of this term should be the proposition $A \Rightarrow B$ itself. But, this is not possible in the theory $\mathcal{U}$ yet, as the proposition $A \Rightarrow B$ has the type *Prop*, and not the type `TYPE`.

A strict view on the Curry-de Bruijn-Howard correspondence leads to identify *Prop* and `TYPE`, yielding the conclusion that all types, including I, and *Prop* itself, are propositions. A more moderate view leads to the introduction of an embedding *Prf* of propositions into types, mapping each proposition A to the type *Prf* A of its proofs. This view is moderate in

two respects. First, the propositions themselves are not the types of their proofs: if t is a proof of A, then it does not have the type A, but the type $\mathit{Prf}\,A$. Second, this embedding is not surjective. So not all types are types of proofs, in particular I and Prop are not.

So, in the theory $\mathcal{U}$, we declare a constant Prf

$$\mathit{Prf} : \mathit{Prop} \to \mathtt{TYPE} \qquad (\mathit{Prf}\text{-decl})$$

When assigning the type $\mathit{Prop} \to \mathtt{TYPE}$ to the constant Prf, we use the fact that $\lambda\Pi/\equiv$ supports dependent types, that is the possibility to build a family of types $\mathit{Prf}\,x$ parameterized with a variable x of type Prop, where Prop is itself of type $\mathtt{TYPE}$.

According to the Brouwer-Heyting-Kolmogorov interpretation, a proof of $A \Rightarrow A$ is a $\lambda\Pi/\equiv$ term expressing a function mapping proofs of A to proofs of A, so that it can be both built and used as a function. In particular, the identity function $\lambda x : \mathit{Prf}\,A, x$ mapping each proof of A to itself is a proof of $A \Rightarrow A$. According to the Curry-de Bruijn-Howard correspondence, this term should have the type $\mathit{Prf}\,(A \Rightarrow A)$, but it has the type $\mathit{Prf}\,A \to \mathit{Prf}\,A$. So, the types $\mathit{Prf}\,(A \Rightarrow A)$ and $\mathit{Prf}\,A \to \mathit{Prf}\,A$ must be identified. To do so, we use the fact that $\lambda\Pi/\equiv$ allows the declaration of rewriting rules, so that $\mathit{Prf}\,(A \Rightarrow A)$ rewrites to $\mathit{Prf}\,A \to \mathit{Prf}\,A$.

$$\mathit{Prf}\,(x \Rightarrow y) \hookrightarrow \mathit{Prf}\,x \to \mathit{Prf}\,y \qquad (\Rightarrow\text{-red})$$

This rule expresses the meaning of the constant $\Rightarrow$. It is, in $\lambda\Pi/\equiv$, the expression of the Brouwer-Heyting-Kolmogorov interpretation of proofs for implication: a proof of $x \Rightarrow y$ is a function mapping proofs of x to proofs of y. So, in the theory $\mathcal{U}$, the Brouwer-Heyting-Kolmogorov interpretation of proofs for implication is made explicit: it is the rule ($\Rightarrow$-red).

3.5. **Universal quantification.** Unlike implication, the universal quantifier binds a variable. Thus, we express the proposition $\forall z\ A$ as the proposition $\forall\ (\lambda z : \mathit{I}, A)$ [Chu40, NM88, Pau93, HHP93], yielding the type $\mathit{I} \to \mathit{Prop}$ for the argument of $\forall$, hence the type $(\mathit{I} \to \mathit{Prop}) \to \mathit{Prop}$ for the constant $\forall$ itself.

But, in the theory $\mathcal{U}$, we allow quantification, not only over the variables of type I, but over variables of any type of object-terms. We could introduce a different quantifier for each type of object-terms, for instance two quantifiers of type $(\mathit{I}_1 \to \mathit{Prop}) \to \mathit{Prop}$ and $(\mathit{I}_2 \to \mathit{Prop}) \to \mathit{Prop}$ if we had two types I_1 and I_2 of object-terms. But, as in some cases, we will have an infinite number of types of object-terms, this would require the introduction of an infinite number of constants.

Thus, we rather want to have a single generic quantifier. But we cannot give the type $\Pi X : \mathtt{TYPE}, (X \to \mathit{Prop}) \to \mathit{Prop}$ to this quantifier, first because in $\lambda\Pi/\equiv$ there is no way to quantifiy over a variable of type $\mathtt{TYPE}$, but also because this would introduce the possibility to quantify over Prop and all the types of the form $\mathit{Prf}\,A$, while we do not always want to consider these types as types of object-terms.

Therefore, in the theory $\mathcal{U}$, we declare a constant Set of type $\mathtt{TYPE}$ for the types of object-terms

$$\mathit{Set} : \mathtt{TYPE} \qquad (\mathit{Set}\text{-decl})$$

a constant ι of type Set

$$\iota : \mathit{Set} \qquad (\iota\text{-decl})$$

a constant El to embed the terms of type Set into terms of type $\mathtt{TYPE}$

$$El : Set \to \texttt{TYPE} \qquad (El\text{-decl})$$

and a rule that reduces the term $El\ \iota$ to I

$$El\ \iota \ \hookrightarrow\ I \qquad (\iota\text{-red})$$

The types of object-terms then have the form $El\ A$ and are distinguished among the other terms of type TYPE.

We can now give the type $\Pi x : Set, (El\ x \to Prop) \to Prop$ to the generic universal quantifier

$$\forall : \Pi x : Set, (El\ x \to Prop) \to Prop \qquad (\forall\text{-decl})$$

and write $\forall\ \iota\ (\lambda z : I, A)$ for the proposition $\forall z\ A$.

Just like for the implication, we declare a rewriting rule expressing that the type of the proofs of the proposition $\forall\ x\ p$ is the type of functions mapping each z of type $El\ x$ to a proof of $p\ z$

$$Prf\ (\forall\ x\ p) \ \hookrightarrow\ \Pi z : El\ x, Prf\ (p\ z) \qquad (\forall\text{-red})$$

Again, the Brouwer-Heyting-Kolmogorov interpretation of proofs for the universal quantifier is made explicit: it is this rule ($\forall$-red).

3.6. **Other constructive connectives and quantifiers.** The other connectives and quantifiers are defined *à la* Russell. For the conjunction, for example, $Prf\ (x \wedge y)$ is defined as $\Pi z : Prop, (Prf\ x \to Prf\ y \to Prf\ z) \to Prf\ z$. This definition does not use the quantifier $\forall$ of the theory $\mathcal{U}$ (so far, in the theory $\mathcal{U}$, we can quantify over the type I, but not over the type $Prop$), but the quantifier Π of the logical framework $\lambda\Pi/\equiv$ itself.

Remark that, *per se*, the quantification on the variable z of type $Prop$ is predicative, as the term $\Pi z : Prop, (Prf\ x \to Prf\ y \to Prf\ z) \to Prf\ z$ has type TYPE and not $Prop$. But, the rule rewriting $Prf\ (x \wedge y)$ to $\Pi z : Prop, (Prf\ x \to Prf\ y \to Prf\ z) \to Prf\ z$ introduces some impredicativity, as the term $x \wedge y$ of type $Prop$ is "defined" as the inverse image, for the embedding Prf, of the type $\Pi z : Prop, (Prf\ x \to Prf\ y \to Prf\ z) \to Prf\ z$, that contains a quantification on a variable of type $Prop$

$$\top : Prop \qquad (\top\text{-decl})$$
$$Prf\ \top \ \hookrightarrow\ \Pi z : Prop, Prf\ z \to Prf\ z \qquad (\top\text{-red})$$
$$\bot : Prop \qquad (\bot\text{-decl})$$
$$Prf\ \bot \ \hookrightarrow\ \Pi z : Prop, Prf\ z \qquad (\bot\text{-red})$$
$$\neg : Prop \to Prop \qquad (\neg\text{-decl})$$
$$Prf\ (\neg\ x) \ \hookrightarrow\ Prf\ x \to \Pi z : Prop, Prf\ z \qquad (\neg\text{-red})$$
$$\wedge : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\wedge\text{-decl})$$
$$Prf\ (x \wedge y) \ \hookrightarrow\ \Pi z : Prop, (Prf\ x \to Prf\ y \to Prf\ z) \to Prf\ z \qquad (\wedge\text{-red})$$
$$\vee : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\vee\text{-decl})$$
$$Prf\ (x \vee y) \ \hookrightarrow\ \Pi z : Prop, (Prf\ x \to Prf\ z) \to (Prf\ y \to Prf\ z) \to Prf\ z \qquad (\vee\text{-red})$$
$$\exists : \Pi a : Set, (El\ a \to Prop) \to Prop \qquad (\exists\text{-decl})$$
$$Prf\ (\exists\ a\ p) \ \hookrightarrow\ \Pi z : Prop, (\Pi x : El\ a, Prf\ (p\ x) \to Prf\ z) \to Prf\ z \qquad (\exists\text{-red})$$

3.7. **Infinity.** Now that we have the symbols $\top$ and $\bot$, we can express that the type I is infinite, that is, that there exists a non-surjective injection from this type to itself. We call this non-surjective injection *succ*. To express its injectivity, we introduce its left inverse *pred*. To express its non-surjectivity, we introduce an element 0, that is not in its image *positive* [DW05]. This choice of notation enables the definition of natural numbers as some elements of type I

$$0 : I \qquad (0\text{-decl})$$
$$succ : I \to I \qquad (succ\text{-decl})$$
$$pred : I \to I \qquad (pred\text{-decl})$$
$$pred\ 0 \hookrightarrow 0 \qquad (pred\text{-red1})$$
$$pred\ (succ\ x) \hookrightarrow x \qquad (pred\text{-red2})$$
$$positive : I \to Prop \qquad (positive\text{-decl})$$
$$positive\ 0 \hookrightarrow \bot \qquad (positive\text{-red1})$$
$$positive\ (succ\ x) \hookrightarrow \top \qquad (positive\text{-red2})$$

3.8. **Classical connectives and quantifiers.** The disjunction in constructive logic and in classical logic are governed by different deduction rules. As the deduction rules express the meaning of the connectives and quantifiers, we can conclude that the disjunction in constructive logic and in classical logic have different meanings. If these disjunctions have different meanings, they should be expressed with different symbols, for instance $\vee$ for the constructive disjunction and $\vee_c$ for the classical one, just like, in classical logic, we use two different symbols for the inclusive disjunction and the exclusive one.

The constructive and the classical disjunction need not belong to different languages, but they can coexist in the same. Ecumenical logics [Pra15, Dow15, PR17, Gri19] are logics where the constructive and classical connectives and quantifiers coexist. A proposition whose connectors and quantifiers are all constructive, is said to be "purely constructive", and one whose connectors and quantifiers are all classical, is said to be "purely classical". The others are said to be "mixed propositions". Any deductive system, where a purely constructive proposition is provable if and only if it is provable in Constructive predicate logic, and where a purely classical proposition is provable if and only if it is provable in Classical predicate logic, is Ecumenical. Ecumenical logics may of course differ on mixed propositions.

Many Ecumenical logics consider the constructive connectives and quantifiers as primitive and attempt to define the classical ones from them, using the negative translation as a definition. There are several options: the classical disjunction, for instance, can be defined in any of the following ways:

(1) $A \vee_c B = \neg\neg(A \vee B)$
(2) $A \vee_c B = (\neg\neg A) \vee (\neg\neg B)$
(3) $A \vee_c B = \neg\neg((\neg\neg A) \vee (\neg\neg B))$

and similarly for the other connectives and quantifiers.

Using these definitions, the proposition $(P \wedge_c Q) \Rightarrow_c P$ is then:

(1) $\neg\neg((\neg\neg(P \wedge Q)) \Rightarrow P)$
(2) $(\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P)$
(3) $\neg\neg((\neg\neg\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P))$

None of them is exactly the negative translation of $(P \wedge Q) \Rightarrow P$ that is

$$\neg\neg((\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P))$$

With Definition (1), the double negations on atomic propositions are missing. This can be repaired in two ways. Predicate symbols of the language can be duplicated [Pra15] into a constructive and a classical counterpart, the latter being the the double negation of the former. Or the syntax of predicate logic can be modified [Gil18]. First, terms are defined. Then, atoms are defined as terms of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol and $t_1, \ldots, t_n$ are terms. Finally, propositions are defined as either explicitly embedded atoms, conjunctions of two propositions, etc. Atoms can be constructively embedded into propositions with the symbol $\triangleright$ or classically with a double-negation version of $\triangleright$. This way, the proposition above is now written $(\triangleright_c P \wedge_c \triangleright_c Q) \Rightarrow_c \triangleright_c P$, which is, by definition, equal to $\neg\neg((\neg\neg((\neg\neg \triangleright P) \wedge (\neg\neg \triangleright Q))) \Rightarrow (\neg\neg \triangleright P))$.

With Definition (2) [AH14], the double negation at the root of the proposition is missing. This again can be repaired [Gri19] by modifying the syntax of Predicate logic, defining first terms, then pre-propositions, that are defined like the propositions in Predicate logic and then propositions, a proposition being obtained by applying a symbol $\circ_c$ to a pre-proposition. Again, this symbol has also a classical version defined as the double negation of the constructive one. This way, the proposition above is written $\circ_c((P \wedge_c Q) \Rightarrow_c P)$ and this proposition is, by definition, equal to $\neg\neg((\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P))$.

Definition (3) [Dow15] is closer to the negative translation except that, in some places, the two negations are replaced with four. But, as $\neg\neg\neg A$ is equivalent to $\neg A$, these extra negations can be removed. Yet, a classical atomic proposition P is the same as its constructive version, while its negative translation is $\neg\neg P$, and in (1) P_c or $\triangleright_c P$ is equal to $\neg\neg P$, as well as $\circ_c P$ in (2). As atomic propositions are not provable anyway, this does not affect provability. But it affects hypothetical provability, leading to duplicate the notion of entailment.

In the theory $\mathcal{U}$, we use Definition (2). Indeed, as we already have a distinction between the proposition A and the type $\mathit{Prf}\ A$ of its proofs, we can just include the symbol $\circ_c$ into the constant Prf introducing a classical version Prf_c of this constant

$$\mathit{Prf}_c : \mathit{Prop} \to \mathtt{TYPE} \qquad (\mathit{Prf}_c\text{-decl})$$
$$\mathit{Prf}_c \hookrightarrow \lambda x : \mathit{Prop}, \mathit{Prf}\ (\neg\neg x) \qquad (\mathit{Prf}_c\text{-red})$$

We can then define the classical connectives and quantifiers as follows

$$\Rightarrow_c : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop} \quad \text{(written infix)} \qquad (\Rightarrow_c\text{-decl})$$
$$\Rightarrow_c \hookrightarrow \lambda x : \mathit{Prop}, \lambda y : \mathit{Prop}, (\neg\neg x) \Rightarrow (\neg\neg y) \qquad (\Rightarrow_c\text{-red})$$
$$\wedge_c : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop} \quad \text{(written infix)} \qquad (\wedge_c\text{-decl})$$
$$\wedge_c \hookrightarrow \lambda x : \mathit{Prop}, \lambda y : \mathit{Prop}, (\neg\neg x) \wedge (\neg\neg y) \qquad (\wedge_c\text{-red})$$
$$\vee_c : \mathit{Prop} \to \mathit{Prop} \to \mathit{Prop} \quad \text{(written infix)} \qquad (\vee_c\text{-decl})$$
$$\vee_c \hookrightarrow \lambda x : \mathit{Prop}, \lambda y : \mathit{Prop}, (\neg\neg x) \vee (\neg\neg y) \qquad (\vee_c\text{-red})$$
$$\forall_c : \Pi a : \mathit{Set}, (\mathit{El}\ a \to \mathit{Prop}) \to \mathit{Prop} \qquad (\forall_c\text{-decl})$$
$$\forall_c \hookrightarrow \lambda a : \mathit{Set}, \lambda p : (\mathit{El}\ a \to \mathit{Prop}), \forall\ a\ (\lambda x : \mathit{El}\ a, \neg\neg(p\ x)) \qquad (\forall_c\text{-red})$$
$$\exists_c : \Pi a : \mathit{Set}, (\mathit{El}\ a \to \mathit{Prop}) \to \mathit{Prop} \qquad (\exists_c\text{-decl})$$
$$\exists_c \hookrightarrow \lambda a : \mathit{Set}, \lambda p : (\mathit{El}\ a \to \mathit{Prop}), \exists\ a\ (\lambda x : \mathit{El}\ a, \neg\neg(p\ x)) \qquad (\exists_c\text{-red})$$

Note that $\top_c$ and $\bot_c$ are $\top$ and $\bot$, by definition. Note also that $\neg\neg\neg A$ is equivalent to $\neg A$, so we do not need to duplicate negation either.

3.9. **Propositions as objects.** So far, we have mainly reconstructed the Predicate logic notions of object-term, proposition, and proof. We can now turn to two notions coming from Simple type theory: propositions as objects and functionality.

Simple type theory is often presented as an independent system, but it can be expressed in several logical frameworks, such as Predicate logic, Isabelle, Deduction modulo theory, Pure type systems, and also $\lambda\Pi/\equiv$. Yet, the relation between Predicate logic and Simple type theory is complex because

- Simple type theory can be expressed in Predicate logic
- and Predicate logic is a restriction of Simple type theory, allowing quantification on variables of type ι only.

So, in Predicate logic, we can express Simple type theory, that contains, as a restriction, Predicate logic, in which we can express Simple type theory, that contains, as a restriction, Predicate logic, in which we can express Simple type theory, that contains, etc. Stacking encodings in this way leads to nonsensical expressions of Simple type theory. But this remark shows that, after having reconstructed Predicate logic in $\lambda\Pi/\equiv$, we have a choice: we can either express Simple type theory in Predicate logic, that is itself expressed in $\lambda\Pi/\equiv$, or express Simple type theory directly in $\lambda\Pi/\equiv$, letting Predicate logic be, a posteriori, a restriction of it, that is, build Simple type theory, not in Predicate logic, but as an extension of Predicate logic.

In the theory $\mathcal{U}$, we choose the second option that leads to a simpler expression of Simple type theory, avoiding the stacking of two encodings. Simple type theory is thus expressed by adding two axioms on top of Predicate logic: one for propositions as objects and one for functionality.

Let us start with propositions as objects. So far, the term ι is the only closed term of type Set. So, we can only quantify over the variables of type $\mathit{El}\ \iota$, that is I. In particular, we cannot quantify over propositions. To do so, we just need to declare a constant o of type Set

$$o : \mathit{Set} \qquad (o\text{-decl})$$

and a rule identifying $\mathit{El}\ o$ and Prop

$$\mathit{El}\ o \hookrightarrow \mathit{Prop} \qquad (o\text{-red})$$

Note that just like there are no terms of type ι, but terms, such as 0, which have type $\mathit{El}\ \iota$, that is I, there are no terms of type o, but terms, such as $\top$, which have type $\mathit{El}\ o$, that is Prop.

Applying the constant $\forall$ to the constant o, we obtain a term of type $(\mathit{El}\ o \to \mathit{Prop}) \to \mathit{Prop}$, that is $(\mathit{Prop} \to \mathit{Prop}) \to \mathit{Prop}$, and we can express the proposition $\forall p\ (p \Rightarrow p)$ as $\forall\ o\ (\lambda p : \mathit{Prop}, p \Rightarrow p)$. The type $\mathit{Prf}\ (\forall\ o\ (\lambda p : \mathit{Prop}, p \Rightarrow p))$ of the proofs of this proposition rewrites to $\Pi p : \mathit{Prop}, \mathit{Prf}\ p \to \mathit{Prf}\ p$. So, the term $\lambda p : \mathit{Prop}, \lambda x : \mathit{Prf}\ p, x$ is a proof of this proposition.

3.10. **Functionality.** Besides ι and o, we introduce more types in the theory, for functions and sets. To do so, we declare a constant

$$\rightsquigarrow : \mathit{Set} \to \mathit{Set} \to \mathit{Set} \quad \text{(written infix)} \qquad (\rightsquigarrow\text{-decl})$$

and a rewriting rule

$$El\ (x \rightsquigarrow y) \ \hookrightarrow\ El\ x \to El\ y \qquad (\rightsquigarrow\text{-red})$$

For instance, these rules enable the construction of the $\lambda\Pi/\equiv$ term $\iota \rightsquigarrow \iota$ of type Set that expresses the simple type $\iota \to \iota$. The $\lambda\Pi/\equiv$ term $El\ (\iota \rightsquigarrow \iota)$ of type `TYPE` rewrites to $I \to I$. The simply typed term $\lambda x : \iota, x$ of type $\iota \to \iota$ is then expressed as the term $\lambda x : I, x$ of type $I \to I$ that is, $El\ (\iota \rightsquigarrow \iota)$.

3.11. **Dependent arrow.** The axiom $(\rightsquigarrow)$ enables us to give simple types to the object-terms expressing functions. We can also give them dependent types, with the dependent versions of this axiom

$$\rightsquigarrow_d : \Pi x : Set, (El\ x \to Set) \to Set \quad \text{(written infix)} \qquad (\rightsquigarrow_d\text{-decl})$$
$$El\ (x \rightsquigarrow_d y) \ \hookrightarrow\ \Pi z : El\ x, El\ (y\ z) \qquad (\rightsquigarrow_d\text{-red})$$

Note that, if we apply the constant $\rightsquigarrow_d$ to a term t and a term $\lambda z : El\ t, u$, where the variable z does not occur in u, then $El\ (t \rightsquigarrow_d \lambda z : El\ t, u)$ rewrites to $El\ t \to El\ u$, just like $El\ (t \rightsquigarrow u)$. Thus, the constant $\rightsquigarrow_d$ is useful only if we can build a term $\lambda z : El\ t, u$ where the variable z occurs in u. With the symbols we have introduced so far, this is not possible. The only constants that can be used to build a term of type Set are ι, o, $\rightsquigarrow$, and $\rightsquigarrow_d$, and the variable z cannot occur free in a term built from these four constants and a variable z of type $El\ t$.

Just like we have a constant ι of type Set, we could add a constant $array$ of type $I \to Set$ such that $array\ n$ is the type of arrays of length n. We could then construct the term $(\iota \rightsquigarrow_d \lambda n : I, array\ n)$ of type Set. Then, the type $El\ (\iota \rightsquigarrow_d \lambda n : I, array\ n)$ that rewrites to $\Pi n : I, El\ (array\ n)$, would be the type of functions mapping a natural number n to an array of length n.

So, this symbol $\rightsquigarrow_d$ becomes useful, only if we add such a constant $array$, object-level dependent types, or the symbols π or $psub$ below.

3.12. **Dependent implication.** In the same way, we can add a dependent implication, where, in the proposition $A \Rightarrow B$, the proof of A may occur in B

$$\Rightarrow_d : \Pi x : Prop, (Prf\ x \to Prop) \to Prop \quad \text{(written infix)} \qquad (\Rightarrow_d\text{-decl})$$
$$Prf\ (x \Rightarrow_d y) \ \hookrightarrow\ \Pi z : Prf\ x, Prf\ (y\ z) \qquad (\Rightarrow_d\text{-red})$$

3.13. **Proofs in object-terms.** To construct an object-term, we sometimes want to apply a function symbol to other object-terms and also to proofs. For instance, we may want to apply the Euclidean division div to two numbers t and u and to a proof that u is positive.

We would like the type of div to be something like

$$El\ (\iota \rightsquigarrow \iota \rightsquigarrow_d \lambda y : I,\ (positive\ y \rightsquigarrow \iota))$$

But the term $(positive\ y \rightsquigarrow \iota)$ is not well typed, as the constant $\rightsquigarrow$ expects, as a first argument, a term of type Set and not of type $Prop$, that is, a type of object-terms and not of proofs.

Thus, we must declare another constant

$$\pi : Prop \to Set \to Set$$

and a rewriting rule

$$El\ (\pi\ x\ y) \ \hookrightarrow\ (Prf\ x) \to (El\ y)$$

Just like for the constant $\rightsquigarrow$ and $\Rightarrow$, we can also have a dependent version of this constant. In fact, in the theory $\mathcal{U}$, we only have this dependent version

$$\pi : \Pi x : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Set}) \to \mathit{Set} \qquad (\pi\text{-decl})$$
$$\mathit{El}\ (\pi\ x\ y) \ \hookrightarrow\ \Pi z : \mathit{Prf}\ x, \mathit{El}\ (y\ z) \qquad (\pi\text{-red})$$

This way, we can give, to the constant div, the type

$$\mathit{El}\ (\iota \rightsquigarrow \iota \rightsquigarrow_d \lambda y : I, \pi\ (\mathit{positive}\ y)\ (\lambda z : \mathit{Prf}\ (\mathit{positive}\ y), \iota))$$

that is

$$I \to \Pi y : I, \mathit{Prf}\ (\mathit{positive}\ y) \to I$$

In the same way, if we add a symbol $=$ of type $\Pi x : \mathit{Set}, \mathit{El}\ x \to \mathit{El}\ x \to \mathit{Prop}$, we can express the proposition

$$\mathit{positive}\ y \Rightarrow_d \lambda p : \mathit{Prf}\ (\mathit{positive}\ y), (=\ \iota\ (\mathit{div}\ x\ y\ p)\ (\mathit{div}\ x\ y\ p))$$

enlightening the meaning of the proposition usually written

$$y > 0 \Rightarrow x/y = x/y$$

The proposition $x/y = x/y$ is well-formed, but it contains, besides x and y, an implicit free variable p, for a proof of $y > 0$. This variable is bound by the implication, that needs therefore to be a dependent implication. Hence, the only free variables in $y > 0 \Rightarrow x/y = x/y$ are x and y.

3.14. **Proof irrelevance.** If p and q are two non convertible proofs of the proposition $\mathit{positive}\ 2$, the terms $\mathit{div}\ 7\ 2\ p$ and $\mathit{div}\ 7\ 2\ q$ are not convertible. As a consequence, the proposition

$$=\ \iota\ (\mathit{div}\ 7\ 2\ p)\ (\mathit{div}\ 7\ 2\ q)$$

would not be provable.

To make these terms convertible, we embed the theory into an extended one, that contains another constant

$$\mathit{div}^\dagger : \mathit{El}\ (\iota \rightsquigarrow \iota \rightsquigarrow \iota)$$

and a rule

$$\mathit{div}\ x\ y\ p \ \hookrightarrow\ \mathit{div}^\dagger\ x\ y$$

and we define convertibility in this extended theory. This way, the terms $\mathit{div}\ 7\ 2\ p$ and $\mathit{div}\ 7\ 2\ q$ are convertible, as they both reduce to $\mathit{div}^\dagger\ 7\ 2$.

Note that, in the extended theory, the constant $\mathit{div}^\dagger$ enables the construction of the erroneous term $\mathit{div}^\dagger\ 1\ 0$. But the extended theory is only used to define the convertibility in the restricted one and this term is not a term of the restricted theory. It is not even the reduct of a term of the form $\mathit{div}\ 1\ 0\ r$ [FT19, BH21].

3.15. **Dependent pairs and predicate subtyping.** Instead of declaring a constant *div* that takes three arguments: a number t, a number u, and a proof p that u is positive, we can declare a constant that takes two arguments: a number t and a pair $\mathit{pair}\ \iota\ \mathit{positive}\ u\ p$ formed with a number u and a proof p that u is positive.

The type of the pair $\mathit{pair}\ \iota\ \mathit{positive}\ u\ p$ whose first element is a number and the second a proof that this number is positive is written $\mathit{psub}\ \iota\ \mathit{positive}$, or informally $\{x : \iota \mid \mathit{positive}\ x\}$. It can be called "the type of positive numbers", especially if the pair is proof-irrelevant in its second argument. It is a subtype of the type of natural numbers defined with the predicate *positive*. Therefore, the symbol *psub* introduces predicate subtyping.

We thus declare a constant *psub* and a constant *pair*

$$\mathit{psub} : \Pi t : \mathit{Set}, (\mathit{El}\ t \to \mathit{Prop}) \to \mathit{Set} \qquad (\mathit{psub}\text{-decl})$$
$$\mathit{pair} : \Pi t : \mathit{Set}, \Pi p : \mathit{El}\ t \to \mathit{Prop}, \Pi m : \mathit{El}\ t, \mathit{Prf}\ (p\ m) \to \mathit{El}\ (\mathit{psub}\ t\ p) \qquad (\mathit{pair}\text{-decl})$$

This way, instead of giving the type $\mathit{El}\ (\iota \rightsquigarrow \iota \rightsquigarrow_d \lambda y : \mathit{Prf}\ (\mathit{positive}\ y), \iota)$ to the constant *div*, we can give it the type $\mathit{El}\ (\iota \rightsquigarrow \mathit{psub}\ \iota\ \mathit{positive} \rightsquigarrow \iota)$.

To avoid introducing a new positive number $\mathit{pair}\ \iota\ \mathit{positive}\ 3\ p$ with each proof p that 3 is positive, we make this symbol *pair* proof irrelevant by introducing a symbol $\mathit{pair}^\dagger$ and a rewriting rule that discards the proof

$$\mathit{pair}^\dagger : \Pi t : \mathit{Set}, \Pi p : \mathit{El}\ t \to \mathit{Prop}, \mathit{El}\ t \to \mathit{El}\ (\mathit{psub}\ t\ p) \qquad (\mathit{pair}^\dagger\text{-decl})$$
$$\mathit{pair}\ t\ p\ m\ h \ \hookrightarrow\ \mathit{pair}^\dagger\ t\ p\ m \qquad (\mathit{pair}\text{-red})$$

This declaration and this rewriting rule are not part of the theory $\mathcal{U}$ but of the theory $\mathcal{U}^\dagger$ used to define the conversion on the terms of $\mathcal{U}$.

Finally, we declare the projections *fst* and *snd* together with an associated rewriting rule

$$\mathit{fst} : \Pi t : \mathit{Set}, \Pi p : \mathit{El}\ t \to \mathit{Prop}, \mathit{El}\ (\mathit{psub}\ t\ p) \to \mathit{El}\ t \qquad (\mathit{fst}\text{-decl})$$
$$\mathit{fst}\ t\ p\ (\mathit{pair}^\dagger\ t'\ p'\ m) \ \hookrightarrow\ m \qquad (\mathit{fst}\text{-red})$$
$$\mathit{snd} : \Pi t : \mathit{Set}, \Pi p : \mathit{El}\ t \to \mathit{Prop}, \Pi m : \mathit{El}\ (\mathit{psub}\ t\ p), \mathit{Prf}\ (p\ (\mathit{fst}\ t\ p\ m)) \qquad (\mathit{snd}\text{-decl})$$

Note that the left hand side of the rule (*fst*-red) is not well-typed, but it can match a well-typed term $\mathit{fst}\ A\ B\ (\mathit{pair}^\dagger\ A\ B\ m)$. Yet, we prefer this rule to the non linear one $\mathit{fst}\ t\ p\ (\mathit{pair}^\dagger\ t\ p\ m) \ \hookrightarrow\ m$ that would make confluence proofs more difficult.

Note that there is no rewriting rule for the second projection as the second element of pairs is discarded during rewriting.

3.16. **Dependent types.** When we have the axioms (*El*), (ι), and ($\rightsquigarrow$), the type $I \to I$ of the term *succ* is equivalent to the type $\mathit{El}\ (\iota \rightsquigarrow \iota)$. Hence, this type $I \to I$ is in the image of the embedding *El*. The symbol *array* introduced in Section 3.11 has the type $I \to \mathit{Set}$ and similarly if we have a predicate symbol $\leq$: $\mathit{El}\ (\iota \rightsquigarrow \iota \rightsquigarrow o)$, the term $\lambda n : I, \mathit{psub}\ \iota\ (\lambda m : I, m \leq n)$ also has the type $I \to \mathit{Set}$. Unlike the type $I \to I$ the type $I \to \mathit{Set}$ is not in the image of any embedding. It is well-formed in the framework but not in the theory itself.

To make this type an element of the image of an embedding we can introduce dependent types at the level of objects-terms. To do so, we introduce a constant *Set1* of type `TYPE` and a constant *set* of type *Set1*

$$\mathit{Set1} : \texttt{TYPE} \qquad (\mathit{Set1}\text{-decl})$$
$$\mathit{set} : \mathit{Set1} \qquad (\mathit{set}\text{-decl})$$

a new arrow

$$\rightsquigarrow_d : \Pi x : \mathit{Set}, (\mathit{El}\ x \to \mathit{Set1}) \to \mathit{Set1} \qquad (\rightsquigarrow_d\text{-decl})$$

and an embedding of *Set1* into TYPE, similar to the embeddings *Prf* and *El*

$$\mathit{Ty} : \mathit{Set1} \to \mathtt{TYPE} \qquad (\mathit{Ty}\text{-decl})$$

and we identify $\mathit{Ty}\ \mathit{set}$ with *Set*—like $\mathit{El}\ o$ is identified with *Prop*—and the dependent arrow $\rightsquigarrow_d$ with a product type

$$\mathit{Ty}\ \mathit{set} \hookrightarrow \mathit{Set} \qquad (\mathit{set}\text{-red})$$
$$\mathit{Ty}\ (x \rightsquigarrow_d y) \hookrightarrow \Pi z : \mathit{El}\ x, \mathit{Ty}\ (y\ z) \qquad (\rightsquigarrow_d\text{-red})$$

The type $I \to \mathit{Set}$, is now equivalent to $\mathit{Ty}\ (\iota \rightsquigarrow_d (\lambda n : I, \mathit{set}))$ and is thus in the image of the embedding *Ty*. One could think in simply taking $\mathit{set} : \mathit{Set}$, saving constants *Set1*, *Ty* and $\rightsquigarrow_d$ and their rewrite rules. However, such a declaration would encode the product $(\triangle, \square, \square)$ of system λU^- which is inconsistent [Coq86, Hur95].

3.17. **Prenex predicative type quantification in types.** Using the symbols of the theory $\mathcal{U}$ introduced so far, the symbol for equality $=$ has the type $\Pi x : \mathit{Set}, \mathit{El}\ x \to \mathit{El}\ x \to \mathit{Prop}$ which is not a type of object terms. This motivates the introduction of object-level polymorphism [Gir72, Rey74]. However extending Simple type theory with object-level polymorphism makes it inconsistent [Hur95, Coq86], and similarly it makes the theory $\mathcal{U}$ inconsistent. So, object-level polymorphism in $\mathcal{U}$ is restricted to prenex polymorphism.

To do so, we introduce a new constant *Scheme* of type TYPE

$$\mathit{Scheme} : \mathtt{TYPE} \qquad (\mathit{Scheme}\text{-decl})$$

a constant *Els* to embed the terms of type *Scheme* into terms of type TYPE

$$\mathit{Els} : \mathit{Scheme} \to \mathtt{TYPE} \qquad (\mathit{Els}\text{-decl})$$

a constant $\uparrow$ to embed the terms of type *Set* into terms of type *Scheme* and a rule connecting these embeddings

$$\uparrow : \mathit{Set} \to \mathit{Scheme} \qquad (\uparrow\text{-decl})$$
$$\mathit{Els}\ (\uparrow x) \hookrightarrow \mathit{El}\ x \qquad (\uparrow\text{-red})$$

We then introduce a quantifier for the variables of type *Set* in the terms of type *Scheme* and the associated rewriting rule

$$\forall : (\mathit{Set} \to \mathit{Scheme}) \to \mathit{Scheme} \qquad (\forall\text{-decl})$$
$$\mathit{Els}\ (\forall\ p) \hookrightarrow \Pi x : \mathit{Set}, \mathit{Els}\ (p\ x) \qquad (\forall\text{-red})$$

This way, the type of the identity function is $\mathit{Els}\ (\forall\ (\lambda x : \mathit{Set}, \uparrow (x \rightsquigarrow x)))$. It reduces to $\Pi x : \mathit{Set}, \mathit{El}\ x \to \mathit{El}\ x$. Therefore, it is inhabited by the term $\lambda x : \mathit{Set}, \lambda y : \mathit{El}\ x, y$. In a similar way, the symbol $=$ can then be given the type $\mathit{Els}\ (\forall\ (\lambda x : \mathit{Set}, \uparrow (x \rightsquigarrow x \rightsquigarrow o)))$ that reduces to $\Pi x : \mathit{Set}, \mathit{El}\ x \to \mathit{El}\ x \to \mathit{Prop}$.

3.18. **Prenex predicative type quantification in propositions.** When we express the reflexivity of the polymorphic equality, we need also to quantify over a type variable, but now in a proposition. To be able to do so, we introduce another quantifier and its associated rewriting rule

$$\begin{array}{ll} \mathit{Prop}^{\prime} : (\mathit{Set} \to \mathit{Prop}) \to \mathit{Prop} & (\mathit{Prop}^{\prime}\text{-decl}) \\ \mathit{Prf}\ (\mathit{Prop}^{\prime}\ p) \hookrightarrow \Pi x : \mathit{Set}, \mathit{Prf}\ (p\ x) & (\mathit{Prop}^{\prime}\text{-red}) \end{array}$$

This way, the reflexivity of equality can be expressed as $(\mathit{Prop}^{\prime}\ (\lambda s : \mathit{Set}, \forall\ s\ (\lambda x : \mathit{El}\ s, =\ s\ x\ x)))$.

3.19. **The theory $\mathcal{U}$: putting everything together.** As mentioned in Section 2, we call "axiom" a constant declaration together with its rewrite rules if any. Hence, in the following, we denote by (ι) the axiom consisting of $(\iota\text{-decl})$ and $(\iota\text{-red})$, and similarly for all the other axioms.

The theory $\mathcal{U}$ is then formed with 43 axioms: (I), (Set), (El), (ι), (Prop), (Prf), $(\Rightarrow)$, $(\forall)$, $(\top)$, $(\bot)$, $(\neg)$, $(\wedge)$, $(\vee)$, $(\exists)$, (Prf_c), $(\Rightarrow_c)$, $(\wedge_c)$, $(\vee_c)$, $(\forall_c)$, $(\exists_c)$, (o), $(\rightsquigarrow)$, $(\rightsquigarrow_d)$, $(\Rightarrow_d)$, (π), (0), (succ), (pred), $(\mathit{positive})$, (psub), (pair), $(\mathit{pair}^{\dagger})$, (fst), (snd), $(\mathit{Set1})$, (set), $(\rightsquigarrow_d)$, (Ty), (Scheme), (Els), $(\uparrow)$, $(\forall)$, $(\mathit{Prop}^{\prime})$.

Note that, strictly speaking, the declaration $(\mathit{pair}^{\dagger}\text{-decl})$ and the rule $(\mathit{pair}\text{-red})$ are not part of the theory $\mathcal{U}$, but of its extension $\mathcal{U}^{\dagger}$ used to define the conversion on the terms of $\mathcal{U}$.

Among these axioms, 14 only have a constant declaration, 27 have a constant declaration and one rewriting rule, and 2 have a constant declaration and two rewriting rules. So $\Sigma_{\mathcal{U}}$ contains 43 declarations and $\mathcal{R}_{\mathcal{U}}$ 31 rules.

This large number of axioms is explained by the fact that $\lambda\Pi/\equiv$ is a weaker framework than Predicate logic. The 20 first axioms are needed just to construct notions that are primitive in Predicate logic: terms, propositions, with their 13 constructive and classical connectives and quantifiers, and proofs. So the theory $\mathcal{U}$ is just 23 axioms on top of the definition of Predicate logic.

It is also explained by the fact that axioms are more atomic than in Predicate logic: 4 axioms are needed to express "the" axiom of infinity: (0), (succ), (pred), and $(\mathit{positive})$; 5 to express predicate subtyping: (psub), (pair), $(\mathit{pair}^{\dagger})$, (fst), and (snd); 4 to express dependent types: $(\mathit{Set1})$,(set),$(\rightsquigarrow_d)$,(Ty); and 5 to express prenex polymorphism: (Scheme), (Els), $(\uparrow)$, $(\forall)$, and $(\mathit{Prop}^{\prime})$. The 5 remaining axioms express propositions as objects (o); various forms of functionality: $(\rightsquigarrow)$, $(\rightsquigarrow_d)$, and (π); and dependent implication $(\Rightarrow_d)$.

4. Sub-theories

Not all proofs require all these axioms. Many proofs can be expressed in sub-theories built by bringing together some of the axioms of $\mathcal{U}$, but not all.

Given subsets $\Sigma_{\mathcal{S}}$ of $\Sigma_{\mathcal{U}}$ and $\mathcal{R}_{\mathcal{S}}$ of $\mathcal{R}_{\mathcal{U}}$, we would like to be sure that a proof in $\mathcal{U}$, using only constants in $\Sigma_{\mathcal{S}}$, is a proof in $(\Sigma_{\mathcal{S}}, \mathcal{R}_{\mathcal{S}})$. Such a result is trivial in Predicate logic: for instance, a proof in ZFC which does not use the axiom of choice is a proof in ZF, but it is less straightforward in $\lambda\Pi/\equiv$, because $(\Sigma_{\mathcal{S}}, \mathcal{R}_{\mathcal{S}})$ might not be a theory. So we should not consider any pair $(\Sigma_{\mathcal{S}}, \mathcal{R}_{\mathcal{S}})$. For instance, as Set occurs in the type of El, if we want El in $\Sigma_{\mathcal{S}}$, we must take Set as well. In the same way, as $\mathit{positive}\ (\mathit{succ}\ x)$ rewrites to $\top$, if we want $(\mathit{positive})$ and (succ) in $\Sigma_{\mathcal{S}}$, we must include $\top$ in $\Sigma_{\mathcal{S}}$ and the rule rewriting $\mathit{positive}\ (\mathit{succ}\ x)$ to $\top$ in $\mathcal{R}_{\mathcal{S}}$.

This leads to a definition of a notion of sub-theory and to prove that, if $(\Sigma_1, \mathcal{R}_1)$ is a sub-theory of a theory $(\Sigma_0, \mathcal{R}_0)$, Γ, t and A are in $\Lambda(\Sigma_1)$, and $\Gamma \vdash_{\Sigma_0,\mathcal{R}_0} t : A$, then $\Gamma \vdash_{\Sigma_1,\mathcal{R}_1} t : A$.

This property implies that, if π is a proof of A in $\mathcal{U}$ and both A and π are in $\Lambda(\Sigma_1)$, then π is a proof of A in $(\Sigma_1, \mathcal{R}_1)$, but it does not imply that if A is in $\Lambda(\Sigma_1)$ and A has a proof in $\mathcal{U}$, then it has a proof in $(\Sigma_1, \mathcal{R}_1)$.

4.1. **Fragments.**

Definition 4.1 (Fragment)**.** A signature Σ_1 is included in a signature Σ_0, denoted $\Sigma_1 \subseteq \Sigma_0$, if each declaration $c : A$ of Σ_1 is a declaration of Σ_0.

A system $(\Sigma_1, \mathcal{R}_1)$ is a fragment of a system $(\Sigma_0, \mathcal{R}_0)$, if the following conditions are satisfied:

- $\Sigma_1 \subseteq \Sigma_0$ and $\mathcal{R}_1 \subseteq \mathcal{R}_0$;
- for all $(c : A) \in \Sigma_1$, $\mathit{const}(A) \subseteq |\Sigma_1|$;
- for all $\ell \hookrightarrow r \in \mathcal{R}_0$, if $\mathit{const}(\ell) \subseteq |\Sigma_1|$, then $\mathit{const}(r) \subseteq |\Sigma_1|$ and $\ell \hookrightarrow r \in \mathcal{R}_1$.

We write $\vdash_i$ for $\vdash_{\Sigma_i,\mathcal{R}_i}$, $\hookrightarrow_i$ for $\hookrightarrow_{\beta\mathcal{R}_i}$, and $\equiv_i$ for $\equiv_{\beta\mathcal{R}_i}$.

Lemma 4.2 (Preservation of reduction)**.** *If $(\Sigma_1, \mathcal{R}_1)$ is a fragment of $(\Sigma_0, \mathcal{R}_0)$, $t \in \Lambda(\Sigma_1)$ and $t \hookrightarrow_0 u$, then $t \hookrightarrow_1 u$ and $u \in \Lambda(\Sigma_1)$.*

Proof. By induction on the position where the rule is applied. We only detail the case of a top reduction, the other cases easily follow by induction hypothesis.

So, let $\ell \hookrightarrow r$ be the rule used to rewrite t in u and θ such that $t = \theta\ell$ and $u = \theta r$. As $t \in \Lambda(\Sigma_1)$, we have $\ell \in \Lambda(\Sigma_1)$ and, for all x free in ℓ, $\theta x \in \Lambda(\Sigma_1)$. Thus, as $(\Sigma_1, \mathcal{R}_1)$ is a fragment of $(\Sigma_0, \mathcal{R}_0)$, $r \in \Lambda(\Sigma_1)$ and $\ell \hookrightarrow r \in \mathcal{R}_1$. Therefore $t \hookrightarrow_1 u$ and $u = \theta r \in \Lambda(\Sigma_1)$. □

Lemma 4.3 (Preservation of confluence)**.** *Every fragment of a confluent system is confluent.*

Proof. Let $(\Sigma_1, \mathcal{R}_1)$ be a fragment of a confluent system $(\Sigma_0, \mathcal{R}_0)$. We prove that $\hookrightarrow_1$ is confluent on $\Lambda(\Sigma_1)$. Assume that $t, u, v \in \Lambda(\Sigma_1)$, $t \hookrightarrow_1^* u$ and $t \hookrightarrow_1^* v$. Since $|\Sigma_1| \subseteq |\Sigma_0|$, we have $t, u, v \in \Lambda(\Sigma_0)$. Since $\mathcal{R}_1 \subseteq \mathcal{R}_0$, we have $t \hookrightarrow_0^* u$ and $t \hookrightarrow_0^* v$. By confluence of $\hookrightarrow_0$ on $\Lambda(\Sigma_0)$, there exists a w in $\Lambda(\Sigma_0)$ such that $u \hookrightarrow_0^* w$ and $v \hookrightarrow_0^* w$. Since $u, v \in \Lambda(\Sigma_1)$, by Lemma 4.2, $w \in \Lambda(\Sigma_1)$, $u \hookrightarrow_1^* w$ and $v \hookrightarrow_1^* w$. □

Definition 4.4 (Sub-theory)**.** A system $(\Sigma_1, \mathcal{R}_1)$ is a sub-theory of a theory $(\Sigma_0, \mathcal{R}_0)$, if it is a fragment of $(\Sigma_0, \mathcal{R}_0)$ and it is a theory.

As we already know that $\mathcal{R}_1$ is confluent, this amounts to say that each rule of $\mathcal{R}_1$ preserves typing in $(\Sigma_1, \mathcal{R}_1)$.

4.2. **The fragment theorem.**

Theorem 4.5. *Let $(\Sigma_0, \mathcal{R}_0)$ be a confluent system and $(\Sigma_1, \mathcal{R}_1)$ be a sub-theory of $(\Sigma_0, \mathcal{R}_0)$.*

- *If the judgement $\Gamma \vdash_0 t : D$ is derivable, $\Gamma \in \Lambda(\Sigma_1)$ and $t \in \Lambda(\Sigma_1)$, then there exists $D' \in \Lambda(\Sigma_1)$ such that $D \hookrightarrow_0^* D'$ and the judgement $\Gamma \vdash_1 t : D'$ is derivable.*
- *If the judgement $\vdash_0 \Gamma$ well-formed is derivable and $\Gamma \in \Lambda(\Sigma_1)$, then the judgement $\vdash_1 \Gamma$ well-formed is derivable.*

Proof. By mutual induction on the derivations, and by case analysis on the last typing rule. Before detailing each case, note that the most difficult cases are (abs), (app), and (conv), the other cases are a simple application of the induction hypothesis.

- If the last rule of the derivation is
$$\frac{\Gamma \vdash_0 A : \mathtt{TYPE} \quad \Gamma, x : A \vdash_0 B : s \quad \Gamma, x : A \vdash_0 t : B}{\Gamma \vdash_0 \lambda x : A, t : \Pi x : A, B} \text{ (abs)}$$
as Γ, A, and t are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exists A' in $\Lambda(\Sigma_1)$ such that $\mathtt{TYPE} \hookrightarrow_0^* A'$ and $\Gamma \vdash_1 A : A'$ is derivable, and there exists B' in $\Lambda(\Sigma_1)$ such that $B \hookrightarrow_0^* B'$ and $\Gamma, x : A \vdash_1 t : B'$ is derivable. As $\mathtt{TYPE}$ is a sort, $A' = \mathtt{TYPE}$. Therefore, $\Gamma \vdash_1 A : \mathtt{TYPE}$ is derivable.

  As B is typable and every subterm of a typable term is typable, $\mathtt{KIND}$ does not occur in B. As $B \hookrightarrow_0^* B'$ and no rule contains $\mathtt{KIND}$, $\mathtt{KIND}$ does not occur in B' as well. Hence, $B' \neq \mathtt{KIND}$. By Lemma 2.2, as $\Gamma, x : A \vdash_1 t : B'$ is derivable and $B' \neq \mathtt{KIND}$, there exists a sort s' such that $\Gamma, x : A \vdash_1 B' : s'$ is derivable.

  Thus, by the rule (abs), $\Gamma \vdash_1 \lambda x : A, t : \Pi x : A, B'$ is derivable. So there is $D' = \Pi x : A, B'$ in $\Lambda(\Sigma_1)$ such that $\Pi x : A, B \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 \lambda x : A, t : D'$ is derivable.
- If the last rule of the derivation is
$$\frac{\Gamma \vdash_0 t : \Pi x : A, B \quad \Gamma \vdash_0 u : A}{\Gamma \vdash_0 t\ u : (u/x)B} \text{ (app)}$$
as Γ, t, and u are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exist C and A_2 in $\Lambda(\Sigma_1)$, such that $\Pi x : A, B \hookrightarrow_0^* C$, $\Gamma \vdash_1 t : C$ is derivable, $A \hookrightarrow_0^* A_2$, and $\Gamma \vdash_1 u : A_2$ is derivable. As $\Pi x : A, B \hookrightarrow_0^* C$ and rewriting rules are of the form $(c\ l_1 \dots l_n \hookrightarrow r)$, there exist A_1 and B_1 in $\Lambda(\Sigma_1)$ such that $C = \Pi x : A_1, B_1$, $A \hookrightarrow_0^* A_1$, and $B \hookrightarrow_0^* B_1$. By confluence of $\hookrightarrow_0$, there exists A' such that $A_1 \hookrightarrow_0^* A'$ and $A_2 \hookrightarrow_0^* A'$. By Lemma 4.2, as $A_1 \in \Lambda(\Sigma_1)$ and $A_1 \hookrightarrow_0^* A'$, we have $A' \in \Lambda(\Sigma_1)$ and $A_1 \hookrightarrow_1^* A'$. In a similar way, as $A_2 \in \Lambda(\Sigma_1)$ and $A_2 \hookrightarrow_0^* A'$, we have $A_2 \hookrightarrow_1^* A'$. By Lemma 2.2, as $\Gamma \vdash_1 t : \Pi x : A_1, B_1$ is derivable and $\Pi x : A_1, B_1 \neq \mathtt{KIND}$, there exists a sort s such that $\Gamma \vdash_1 \Pi x : A_1, B_1 : s$ is derivable. Thus, by Lemma 2.2, $\Gamma \vdash_1 A_1 : \mathtt{TYPE}$ is derivable.

  As $\Gamma \vdash_1 \Pi x : A_1, B_1 : s$, $\Pi x : A_1, B_1 \hookrightarrow_1^* \Pi x : A', B_1$, and $(\Sigma_1, \mathcal{R}_1)$ preserves typing, $\Gamma \vdash_1 \Pi x : A', B_1 : s$ is derivable. In a similar way, as $\Gamma \vdash_1 A_1 : \mathtt{TYPE}$ is derivable, and $A_1 \hookrightarrow_1^* A'$, $\Gamma \vdash_1 A' : \mathtt{TYPE}$ is derivable. Therefore, by the rule (conv), $\Gamma \vdash_1 t : \Pi x : A', B_1$ and $\Gamma \vdash_1 u : A'$ are derivable. Therefore, by the rule (app), $\Gamma \vdash_1 t\ u : (u/x)B_1$ is derivable. So there exists $D' = (u/x)B_1$ in $\Lambda(\Sigma_1)$, such that $(u/x)B \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 t\ u : D'$ is derivable.
- If the last rule of the derivation is
$$\frac{\Gamma \vdash_0 t : A \quad \Gamma \vdash_0 B : s}{\Gamma \vdash_0 t : B} \text{ (conv) } A \equiv_{\beta\mathcal{R}_0} B$$
as Γ and t are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exists A' in $\Lambda(\Sigma_1)$ such that $A \hookrightarrow_0^* A'$ and $\Gamma \vdash_1 t : A'$ is derivable. By confluence of $\hookrightarrow_0$, there exists C such that $A' \hookrightarrow_0^* C$ and $B \hookrightarrow_0^* C$. As $A' \in \Lambda(\Sigma_1)$ and $A' \hookrightarrow_0^* C$ we have, by Lemma 4.2, $C \in \Lambda(\Sigma_1)$ and $A' \hookrightarrow_1^* C$.

  As B is typable and every subterm of a typable term is typable, $\mathtt{KIND}$ does not occur in B. As $B \hookrightarrow_0^* C$ and no rule contains $\mathtt{KIND}$, $\mathtt{KIND}$ does not occur in C as well. Thus $C \neq \mathtt{KIND}$. As $A' \hookrightarrow_0^* C$, $A' \neq \mathtt{KIND}$. By Lemma 2.2, as $\Gamma \vdash_1 t : A'$ and $A' \neq \mathtt{KIND}$, there exists a sort s' such that $\Gamma \vdash_1 A' : s'$ is derivable. Thus, as $A' \hookrightarrow_1^* C$, and $(\Sigma_1, \mathcal{R}_1)$

preserves typing, $\Gamma \vdash_1 C : s'$ is derivable. a As $\Gamma \vdash_1 t : A'$ and $\Gamma \vdash_1 C : s'$ are derivable and $A' \hookrightarrow_1 C$, by the rule (conv), $\Gamma \vdash_1 t : C$ is derivable. Thus there exists $D' = C$ in $\Lambda(\Sigma_1)$ such that $\Gamma \vdash_1 t : D'$ is derivable and $B \hookrightarrow_0^* D'$.

- If the last rule of the derivation is

$$\frac{}{\vdash_0 [\,] \text{ well-formed}} \text{ (empty)}$$

by the rule (empty), $\vdash_1 [\,]$ well-formed is derivable.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash_0 A : s}{\vdash_0 \Gamma, x : A \text{ well-formed}} \text{ (decl)}$$

as Γ and A are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exist A' in $\Lambda(\Sigma_1)$ such that $s \hookrightarrow_0^* A'$ and $\Gamma \vdash_1 A : A'$ is derivable. As s is a sort, $A' = s$. Therefore, $\Gamma \vdash_1 A : s$ is derivable and, by the rule (decl), $\vdash_1 \Gamma, x : A$ well-formed is derivable.

- If the last rule of the derivation is

$$\frac{\vdash_0 \Gamma \text{ well-formed}}{\Gamma \vdash_0 \texttt{TYPE} : \texttt{KIND}} \text{ (sort)}$$

as Γ is in $\Lambda(\Sigma_1)$, by induction hypothesis, $\vdash_1 \Gamma$ well-formed is derivable. Thus, by the rule (sort), $\Gamma \vdash_1 \texttt{TYPE} : \texttt{KIND}$ is derivable. So there exists $D' = \texttt{KIND}$ in $\Lambda(\Sigma_1)$ such that $\texttt{KIND} \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 \texttt{TYPE} : D'$.

- If the last rule of the derivation is

$$\frac{\vdash_0 \Gamma \text{ well-formed} \quad \vdash_0 A : s}{\Gamma \vdash_0 c : A} \text{ (const) } c : A \in \Sigma_0$$

as Γ is in $\Lambda(\Sigma_1)$, by induction hypothesis, $\vdash_1 \Gamma$ well-formed is derivable. And as c is in $\Lambda(\Sigma_1)$, it is in $|\Sigma_1|$, thus $c : A$ is in Σ_1 and, since $(\Sigma_1, \mathcal{R}_1)$ is a fragment of $(\Sigma_0, \mathcal{R}_0)$, $A \in \Lambda(\Sigma_1)$.

Thus, by induction hypothesis, there exists A' such that $\vdash_1 A : A'$ is derivable and $s \hookrightarrow_0^* A'$. As s is a sort, $A' = s$. So $\vdash_1 A : s$ is derivable. Thus, by the rule (const), $\Gamma \vdash_1 c : A$ is derivable. So, there exists $D' = A$ in $\Lambda(\Sigma_1)$ such that $A \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 c : D'$ is derivable.

- If the last rule of the derivation is

$$\frac{\vdash_0 \Gamma \text{ well-formed}}{\Gamma \vdash_0 x : A} \text{ (var) } x : A \in \Gamma$$

as Γ is in $\Lambda(\Sigma_1)$, by induction hypothesis, $\vdash_1 \Gamma$ well-formed is derivable. Thus, by the rule (var), $\Gamma \vdash_1 x : A$ is derivable. So there exists $D' = A$ in $\Lambda(\Sigma_1)$ such that $A \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 x : D'$.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash_0 A : \texttt{TYPE} \quad \Gamma, x : A \vdash_0 B : s}{\Gamma \vdash_0 \Pi x : A, B : s} \text{ (prod)}$$

as Γ, A, and B are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exists A' in $\Lambda(\Sigma_1)$ such that $\texttt{TYPE} \hookrightarrow_0^* A'$ and $\Gamma \vdash_1 A : A'$ is derivable and there exists B' in $\Lambda(\Sigma_1)$ such that $s \hookrightarrow_0^* B'$ and $\Gamma, x : A \vdash_1 B : B'$ is derivable. As $\texttt{TYPE}$ and s are sorts, $A' = \texttt{TYPE}$ and $B' = s$. Therefore, $\Gamma \vdash_1 A : \texttt{TYPE}$ and $\Gamma, x : A \vdash_1 B : s$ are derivable. Thus, by the rule (prod), $\Gamma \vdash_1 \Pi x : A, B : s$ is derivable. So there exists $D' = s$ in $\Lambda(\Sigma_1)$ such that $s \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 \Pi x : A, B : D'$ is derivable. □

Corollary 4.6. *Let $(\Sigma_0, \mathcal{R}_0)$ be a confluent system, $(\Sigma_1, \mathcal{R}_1)$ be a sub-theory of $(\Sigma_0, \mathcal{R}_0)$. If $\Gamma \vdash_0 t : D$, $\Gamma \in \Lambda(\Sigma_1)$, $t \in \Lambda(\Sigma_1)$, and $D \in \Lambda(\Sigma_1)$, then $\Gamma \vdash_1 t : D$.*

In particular, if $(\Sigma_0, \mathcal{R}_0)$ is a theory, $(\Sigma_1, \mathcal{R}_1)$ is a sub-theory of $(\Sigma_0, \mathcal{R}_0)$, $\Gamma \vdash_0 t : D$, $\Gamma \in \Lambda(\Sigma_1)$, $t \in \Lambda(\Sigma_1)$, and $D \in \Lambda(\Sigma_1)$, then $\Gamma \vdash_1 t : D$.

Proof. There is a $D' \in \Lambda(\Sigma_1)$ such that $D \hookrightarrow_0^* D'$ and $\Gamma \vdash_1 t : D'$. As $D \in \Lambda(\Sigma_1)$ and $D \hookrightarrow_0^* D'$. By Lemma 4.2 we have $D \hookrightarrow_1^* D'$, and we conclude with the rule (conv). □

Theorem 4.7 (Sub-theories of $\mathcal{U}$)**.** *Every fragment $(\Sigma_1, \mathcal{R}_1)$ of $\mathcal{U}$ (including $\mathcal{U}$ itself) is a theory, that is, is confluent and preserves typing.*

Proof. The relation $\hookrightarrow_{\beta\mathcal{R}_\mathcal{U}}$ is confluent on $\Lambda(\Sigma_\mathcal{U})$ since it is an orthogonal combinatory reduction system [KvOvR93]. Hence, after the fragment theorem, it is sufficient to prove that every rule of $\mathcal{R}_\mathcal{U}$ preserves typing in any fragment $(\Sigma_1, \mathcal{R}_1)$ containing the symbols of the rule.

To this end, we will use the criterion described in [Bla20, Theorem 19] which consists in computing the equations that must be satisfied for a rule left-hand side to be typable, which are system-independent, and then check that the right-hand side has the same type modulo these equations in the desired system: for all rules $l \hookrightarrow r \in \Lambda(\Sigma_1)$, sets of equations $\mathcal{E}$ and terms T, if the inferred type of l is T, the typability constraints of l are $\mathcal{E}$, and r has type type T in the system $\Lambda(\Sigma_1)$ whose conversion relation $\equiv_{\beta\mathcal{R}\mathcal{E}}$ has been enriched with $\mathcal{E}$, then $l \hookrightarrow r$ preserves typing in $\Lambda(\Sigma_1)$.

This criterion can easily be checked for all the rules but (*pred*-red2) and (*fst*-red) because, except in those two cases, the left-hand side and the right-hand side have the same type.

In (*pred*-red2), $\mathit{pred}\ (\mathit{succ}\ x) \hookrightarrow x$, the left-hand side has type I if the equation $\mathit{type}(x) = I$ is satisfied. Modulo this equation, the right-hand side has type I in any fragment containing the symbols of the rule.

In (*fst*-red), $\mathit{fst}\ t\ p\ (\mathit{pair}^\dagger\ t'\ p'\ m) \hookrightarrow m$, the left-hand side has type $\mathit{El}\ t$ if $\mathit{type}(t) = \mathit{Set}$, $\mathit{type}(p) = \mathit{El}\ t \to \mathit{Prop}$, $\mathit{El}\ (\mathit{psub}\ t'\ p') = \mathit{El}\ (\mathit{psub}\ t\ p)$, $\mathit{type}(t') = \mathit{Set}$, $\mathit{type}(p') = \mathit{El}\ t' \to \mathit{Prop}$, and $\mathit{type}(m) = \mathit{El}\ t'$. But, in $\mathcal{U}$, there is no rule of the form $\mathit{El}\ (\mathit{psub}\ t\ p) \hookrightarrow r$. Hence, by confluence, the equation $\mathit{El}\ (\mathit{psub}\ t'\ p') = \mathit{El}\ (\mathit{psub}\ t\ p)$ is equivalent to the equations $t' = t$ and $p' = p$. Therefore, the right-hand side is of type $\mathit{El}\ t$ in every fragment of $\mathcal{U}$ containing the symbols of the rule. □

5. Examples of sub-theories of the theory $\mathcal{U}$

We finally identify 15 sub-theories of the theory $\mathcal{U}$, that correspond to known theories. For each of these sub-theories $(\Sigma_\mathcal{S}, \mathcal{R}_\mathcal{S})$, according to the Corollary 4.6, if Γ, t, and A are in $\Lambda(\Sigma_\mathcal{S})$, and $\Gamma \vdash_{\Sigma_\mathcal{U}, \mathcal{R}_\mathcal{U}} t : A$, then $\Gamma \vdash_{\mathcal{R}_\mathcal{S}, \Sigma_\mathcal{S}} t : A$.

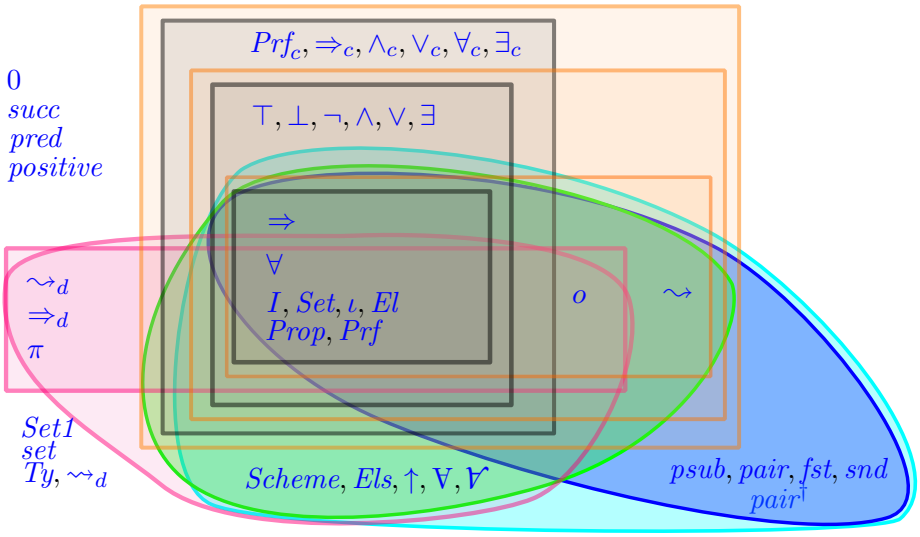

Figure 2: **The wind rose.** In black: Minimal, Constructive, and Ecumenical predicate logic. In orange: Minimal, Constructive, and Ecumenical simple type theory. In green: Simple type theory with prenex polymorphism. In blue: Simple type theory with predicate subtyping. In cyan: Simple type theory with predicate subtyping and prenex polymorphism. In pink: the Calculus of constructions with a constant ι, without and with prenex polymorphism.

5.1. **Minimal predicate logic.** The 8 axioms (I), (Set), (El), (ι), ($Prop$), (Prf), ($\Rightarrow$), and ($\forall$) define Minimal predicate logic $(\Sigma_{\mathcal{M}}, \mathcal{R}_{\mathcal{M}})$.

$$\begin{array}{rcl} I & : & \texttt{TYPE} \\ Set & : & \texttt{TYPE} \\ El & : & Set \to \texttt{TYPE} \\ \iota & : & Set \\ El\ \iota & \hookrightarrow & I \\ Prop & : & \texttt{TYPE} \\ Prf & : & Prop \to \texttt{TYPE} \\ \Rightarrow & : & Prop \to Prop \to Prop \\ Prf\ (x \Rightarrow y) & \hookrightarrow & Prf\ x \to Prf\ y \\ \forall & : & \Pi x : Set, (El\ x \to Prop) \to Prop \\ Prf\ (\forall\ x\ p) & \hookrightarrow & \Pi z : El\ x, Prf\ (p\ z) \end{array}$$

We could save the declaration (I-decl) and the rule (ι-red) by using $El\ \iota$ instead of I.

This theory can be proven equivalent to more common formulations of Minimal predicate logic. To do so, consider a language $\mathcal{L}$ in predicate logic. We define a corresponding $\lambda\Pi/\equiv$ context $\Gamma_{\mathcal{L}}$ containing for each constant f of $\mathcal{L}$ a constant f of type $I \to \ldots \to I \to I$ and for each predicate symbol P of $\mathcal{L}$ a constant P of type $I \to \ldots \to I \to Prop$. A term (resp. a proposition) of minimal predicate logic t of $\mathcal{L}$ translates in the natural way to a $\lambda\Pi/\equiv$ term of type I (resp. $Prop$) in the theory $(\Sigma_{\mathcal{M}}, \mathcal{R}_{\mathcal{M}})$ and in the context $\Gamma_{\mathcal{L}}, \Delta$, where Δ contains, for each variable x free in t, a variable x of type I. We use the same notation for the term (resp. the proposition) and its translation.

Theorem 5.1. *Let $\mathcal{L}$ be a language and $A_1, ..., A_n \vdash B$ be a sequent of minimal predicate logic in $\mathcal{L}$. Let $\Gamma_\mathcal{L}$ containing for each constant f of $\mathcal{L}$ a constant f of type $I \to \ldots \to I \to I$ and for each predicate symbol P of $\mathcal{L}$ a constant P of type $I \to \ldots \to I \to Prop$. Let Δ be a context containing for each variable x free in $A_1, ..., A_n \vdash B$, a variable x of type I. Let Δ' be a context containing, for each hypothesis A_i, a variable a_i of type $Prf\ A_i$.*

Then, the sequent $A_1, ..., A_n \vdash B$ has a proof in minimal logic, if and only if there exists a $\lambda\Pi/\equiv$ term π such that $\Gamma_\mathcal{L}, \Delta, \Delta' \vdash_{\Sigma_\mathcal{M}, \mathcal{R}_\mathcal{M}} \pi : Prf\ B$.

Proof. The left-to-right implication is a trivial induction on the structure of the proof.

For the converse, it is enough to consider an irreducible term π of type $Prf\ B$ since one can prove that $\hookrightarrow_{\beta\mathcal{R}_\mathcal{M}}$ terminates, by applying [BGH19] for instance. We then prove, by induction on π, that the sequent $A_1, ..., A_n \vdash B$ has a proof in minimal logic. As π has the type $Prf\ B$, it is neither a sort, nor a product, thus it is either an abstraction or a term of the form $z\ \rho_1\ ...\ \rho_p$.

- If π is an abstraction then $Prf\ B$ is equivalent to a product. Hence, B either has the form $C \Rightarrow D$ or $\forall\ \iota\ \lambda x : I, D$. In the first case $\pi = \lambda x : Prf\ C, \pi'$ and π' is a term of type $Prf\ D$ in $\Gamma_\mathcal{L}, \Delta, \Delta', x : Prf\ C$. By induction hypothesis, the sequent $A_1, ..., A_n, C \vdash D$ has a proof and so does the sequent $A_1, ..., A_n \vdash C \Rightarrow D$. In the second $\pi = \lambda x : I, \pi'$ and π' is a term of type $Prf\ D$ in $\Gamma_\mathcal{L}, \Delta, x : I, \Delta'$. By induction hypothesis, the sequent $A_1, ..., A_n \vdash D$ has a proof and so does the sequent $A_1, ..., A_n \vdash \forall\ \iota\ \lambda x : I, D$.
- If π has the form $z\ \rho_1\ ...\ \rho_p$, then as it has the type $Prf\ B$, z can neither be a constant of Σ_M, nor a variable of $\Gamma_\mathcal{L}$, Δ. Hence, it is a variable of Δ'. Thus, it has the type $Prf\ A_i$ for some i. We prove, by induction on j that the term $z\ \rho_1\ ...\ \rho_j$ has the type $Prf\ C$ for some proposition C, such that the sequent $A_1, ..., A_n \vdash C$ has a proof. For $j = 0$, the sequent $A_1, ..., A_n \vdash A_i$ has a proof. Assume the property holds for j. Then, as the term $z\ \rho_1\ ...\ \rho_j\ \rho_{j+1}$ is well typed, the type $Prf\ C$ is a product type and C either has the form $D \Rightarrow E$ or $\forall\ \iota\ \lambda x : I, E$. In the first case ρ_{j+1} is a term of type $Prf\ D$, by induction hypothesis, the sequent $A_1, ..., A_n \vdash D$ has a proof, hence the term $z\ \rho_1\ ...\ \rho_j\ \rho_{j+1}$ has the type $Prf\ E$ and the sequent $A_1, ..., A_n \vdash E$ has a proof. In the second case, ρ_{j+1} is an irreducible term of type I, thus it is an object-term, the term $z\ \rho_1\ ...\ \rho_j\ \rho_{j+1}$ has the type $Prf\ (\rho_{j+1}/x)E$, and the sequent $A_1, ..., A_n \vdash (\rho_{j+1}/x)E$ has a proof. □

As Minimal predicate logic is itself a logical framework, it must be complemented with more axioms, such as the axioms of geometry, arithmetic, etc.

5.2. **Constructive predicate logic.** The 14 axioms (I),(Set), (El), (ι), $(Prop)$, (Prf), $(\Rightarrow)$, $(\forall)$, $(\top)$, $(\bot)$, $(\neg)$, $(\wedge)$, $(\vee)$, and $(\exists)$ define Constructive predicate logic. This theory can be proven equivalent to more common formulations of Constructive predicate logic [Dor10, ABC+16].

5.3. **Ecumenical predicate logic.** The 20 axioms (I), (Set), (El), (ι), $(Prop)$, (Prf), $(\Rightarrow)$, $(\forall)$, $(\top)$, $(\bot)$, $(\neg)$, $(\wedge)$, $(\vee)$, $(\exists)$, (Prf_c), $(\Rightarrow_c)$, $(\wedge_c)$, $(\vee_c)$, $(\forall_c)$, and $(\exists_c)$ define Ecumenical predicate logic. This theory can be proven equivalent to more common formulations of Ecumenical predicate logic [Gri21].

Note that classical predicate logic is not a sub-theory of the theory $\mathcal{U}$, because the classical connectives and quantifiers depend on the constructive ones. Yet, it is known that if

a proposition contains only classical connectives and quantifiers, it is provable in Ecumenical predicate logic if and only if it is provable in classical predicate logic.

5.4. **Minimal simple type theory.** Adding the two axioms (o) and ($\rightsquigarrow$) to Predicate logic defines Simple type theory. Indeed, Simple type theory is the theory of propositional contents and functions. A simple type T is naturally translated to $\lambda\Pi/\equiv$ as a term T of type Set, using types ι and o and the arrow construction $\rightsquigarrow$. The higher order terms are shallowly translated: λ-abstractions and applications are translated using respectively $\lambda\Pi/\equiv$'s λ-abstractions and applications.

The 10 axioms (I), (Set), (ι), (El), (Prop), (Prf), ($\Rightarrow$), ($\forall$), (o), and ($\rightsquigarrow$) define Minimal simple type theory. And this theory can be proven equivalent to more common formulations of Minimal simple type theory [Ass15, ABC$^+$16]. Just like we can save the declaration (I-decl) and the rule (ι-red) by replacing everywhere I with $\mathit{El}\,\iota$, we could save the declaration (Prop-decl) and the rule (o-red) by replacing everywhere Prop with $\mathit{El}\,o$. This presentation is the usual presentation of Simple type theory in $\lambda\Pi/\equiv$ [ABC$^+$16] with 8 declarations and 3 reduction rules. However, doing so, the obtained presentation of Simple type theory is not an extension of Minimal predicate logic anymore.

5.5. **Constructive simple type theory.** The 16 axioms (I), (Set), (El), (ι), (Prop), (Prf), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (o) and ($\rightsquigarrow$) define Constructive simple type theory.

5.6. **Ecumenical simple type theory.** The 22 axioms (I), (Set), (El), (ι), (Prop), (Prf), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (Prf_c), ($\Rightarrow_c$), ($\wedge_c$), ($\vee_c$), ($\forall_c$), ($\exists_c$), (o) and ($\rightsquigarrow$) define Ecumenical simple type theory. And this theory can be proven equivalent to more common formulations of Ecumenical simple type theory [Gri21].

5.7. **Simple type theory with predicate subtyping.** Adding to the 10 axioms of Minimal simple type theory the 5 axioms of predicate subtyping yields Minimal simple type theory with predicate subtyping, formed with the 15 axioms (I), (Set), (ι), (El), (Prop), (Prf), ($\Rightarrow$), ($\forall$), (o), ($\rightsquigarrow$), (psub), (pair), ($\mathit{pair}^\dagger$), (fst), and (snd). This theory has been studied by F. Gilbert [Gil18] as a verifiable language for a minimal and idealised version of the type system of the PVS proof assistant. It can be proven equivalent to more common formulations of Minimal simple type theory with predicate subtyping [Gil18, BH21]. Such formulations like PVS [OS97] often use predicate subtyping implicitly to provide a lighter syntax without (pair), ($\mathit{pair}^\dagger$), (fst) nor (snd) but at the expense of losing uniqueness of type and making type-checking undecidable. In these cases, terms generally do not hold the proofs needed to be of a sub-type, which provides proof irrelevance. Our implementation of proof irrelevance of Section 3.14 extends the conversion in order to ignore these proofs.

5.8. **Simple type theory with prenex predicative polymorphism.** Adding to the 10 axioms of Minimal simple type theory the 5 axioms of prenex predicative polymorphism yields Simple type theory with prenex predicative polymorphism (STT$\forall$) [Thi18, Thi20] formed with the 15 axioms (I), (Set), (El), (ι), (Prop), (Prf), ($\Rightarrow$), ($\forall$), (o), ($\rightsquigarrow$), (Scheme), (Els), ($\uparrow$), ($\forall$), and ($\forall'$).

5.9. **Simple type theory with predicate subtyping and prenex polymorphism.** Adding to the 10 axioms of Minimal simple type theory both the 5 axioms of predicate subtyping and the 5 axioms of prenex polymorphism yields a sub-theory with 20 axioms which is a subsystem of PVS [OS97] handling both predicate subtyping and prenex polymorphism.

5.10. **The Calculus of constructions.** Pure type systems [Ber88, Ter89, Bar92] are a family of typed λ-calculi. An example is the $\lambda\Pi$-calculus, the λ-calculus with dependent types, which is at the basis of $\lambda\Pi/\equiv$ itself. As we have seen, in $\lambda\Pi/\equiv$, we have two constants, `TYPE` and `KIND`, `TYPE` has type `KIND`, and we can build a product type $\Pi x : A, B$ when both A and B have type `TYPE`, in which case the product type $\Pi x : A, B$ itself has type `TYPE` or when A has type `TYPE` and B has type `KIND`, in which case the result has type `KIND`.

A Pure type system, in general, is defined with a set of symbols, such as `TYPE` and `KIND`, called "sorts", a set of axioms of the form $\langle s_1, s_2\rangle$, expressing that the sort s_1 has type s_2, for example $\langle\texttt{TYPE},\texttt{KIND}\rangle$, and a set of rules of the form $\langle s_1, s_2, s_3\rangle$, expressing that we can build the product type $\Pi x : A, B$, when A has type s_1 and B has type s_2, and that the product type itself has type s_3, for example $\langle\texttt{TYPE},\texttt{TYPE},\texttt{TYPE}\rangle$ and $\langle\texttt{TYPE},\texttt{KIND},\texttt{KIND}\rangle$. When the set of axioms is functional, each sort has at most one type and when the set of rules is functional, each product type has at most one type. In this case the Pure type system is said to be "functional".

To have more compact notation, we often write $*$ for the sort `TYPE` and $\Box$ for the sort `KIND`. So the $\lambda\Pi$-calculus is defined with the sorts $*$ and $\Box$, the axiom $\langle *, \Box\rangle$, and the rules $\langle *, *, *\rangle$ and $\langle *, \Box, \Box\rangle$. Adding the rules $\langle \Box, *, *\rangle$ and $\langle \Box, \Box, \Box\rangle$ yields the Calculus of constructions [CH88].

All functional Pure type systems can be expressed in $\lambda\Pi/\equiv$ [CD07]: for each sorts s, we introduce two constants U_s of type `TYPE` and ε_s of type $U_s \to \texttt{TYPE}$, for each axiom $\langle s_1, s_2\rangle$, a constant $\dot{s}_1$ of type U_{s_2}, and a reduction rule

$$\varepsilon_{s_2}\ \dot{s}_1 \hookrightarrow U_{s_1}$$

and for each rule $\langle s_1, s_2, s_3\rangle$, a constant $\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}$ of type $\Pi x : U_{s_1}, (\varepsilon_{s_1}\ x \to U_{s_2}) \to U_{s_3}$ and a reduction rule

$$\varepsilon_{s_3}\ (\dot{\Pi}_{\langle s_1,s_2,s_3\rangle}\ x\ y) \hookrightarrow \Pi z : \varepsilon_{s_1}\ x, \varepsilon_{s_2}\ (y\ z)$$

We obtain this way a correct and conservative expression of the Pure type system [CD07, ABC+16]. For instance, the expression of the Calculus of constructions yields 9 declarations and 5 rules. Writing Prop for U_*, Prf for ε_*, Set for $U_\Box$, El for $\varepsilon_\Box$, o for $*$, $\Rightarrow_d$ for $\dot{\Pi}_{\langle *,*,*\rangle}$, $\forall$ for $\dot{\Pi}_{\langle \Box,*,*\rangle}$, π for $\dot{\Pi}_{\langle *,\Box,\Box\rangle}$, and $\rightsquigarrow_d$ for $\dot{\Pi}_{\langle \Box,\Box,\Box\rangle}$ we get exactly the 9 axioms (Prop), (Prf), (Set), (El), (o), ($\Rightarrow_d$), (π), ($\forall$), and ($\rightsquigarrow_d$) and this theory is equivalent to more common formulations of the Calculus of constructions.

As $\Rightarrow_d$ is $\dot{\Pi}_{\langle *,*,*\rangle}$, $\forall$ is $\dot{\Pi}_{\langle \Box,*,*\rangle}$, π is $\dot{\Pi}_{\langle *,\Box,\Box\rangle}$, and $\rightsquigarrow_d$ is $\dot{\Pi}_{\langle \Box,\Box,\Box\rangle}$, using the terminology of Barendregt's λ-cube [Bar92], the axiom (π) expresses dependent types, the axiom ($\forall$) polymorphism, and the axiom ($\rightsquigarrow_d$) type constructors.

Note that these constants have similar types

$$\begin{array}{rcl} \Rightarrow_d & : & \Pi x : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prop}) \to \mathit{Prop} \\ \pi & : & \Pi x : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Set}) \to \mathit{Set} \\ \forall & : & \Pi x : \mathit{Set}, (\mathit{El}\ x \to \mathit{Prop}) \to \mathit{Prop} \\ \rightsquigarrow_d & : & \Pi x : \mathit{Set}, (\mathit{El}\ x \to \mathit{Set}) \to \mathit{Set} \end{array}$$

So if Γ is a context and A is a term in the Calculus of constructions then A is inhabited in Γ in the Calculus of constructions if and only if the translation $\|A\|$ of A in $\lambda\Pi/\equiv$ is inhabited in the translation $\|\Gamma\|$ of Γ in $\lambda\Pi/\equiv$ [CD07, ABC+16]. So, the formulation of the Calculus of constructions in $\lambda\Pi/\equiv$ is a conservative extension of the original formulation of the Calculus of constructions. In the context $\|\Gamma\|$, variables have a $\lambda\Pi/\equiv$ type of the form $\mathit{Prf}\ u$ or $\mathit{El}\ u$, and none of them can have the type Set. However, in $\lambda\Pi/\equiv$, nothing prevents from declaring a variable of type Set. Hence, in $\lambda\Pi/\equiv$, the judgement $x : \mathit{Set} \vdash x : \mathit{Set}$ can be derived, but it is not in the image of the encoding.

5.11. **The Calculus of constructions with variables of type $\Box$.** To allow the declaration of variables of type $\Box$ in the Calculus of constructions, a possibility is to add a sort $\triangle$ and an axiom $\Box : \triangle$ [Geu95], making the sort $\triangle$ a singleton sort that contains only one closed irreducible term: $\Box$.

Expressing this Pure type system in $\lambda\Pi/\equiv$ introduces two declarations for the sort $\triangle$ and one declaration and one rule for the axiom $\Box : \triangle$

- $U_\triangle : \mathtt{TYPE}$
- $\varepsilon_\triangle : U_\triangle \to \mathtt{TYPE}$
- $\boxdot : U_\triangle$
- $\varepsilon_\triangle\ \boxdot \hookrightarrow \mathit{Set}$

and the variables in a context $\|\Gamma\|$ now have the type $\mathit{Prf}\ u$, $\mathit{El}\ u$, or $\varepsilon_\triangle\ u$. Just like U_* is written Prop in $\mathcal{U}$, $U_\triangle$ is written $\mathit{Set1}$, $\varepsilon_\triangle$ is written Ty, and $\boxdot$ is written set.

But this theory can be simplified. Indeed, just like $\Box$ is the only closed irreducible term of type $\triangle$, set is the only closed irreducible term of type $\mathit{Set1}$ and thus for any closed term t of type $\mathit{Set1}$, the term $\mathit{Ty}\ t$ reduces to Set. So, we can replace everywhere the terms of the form $\mathit{Ty}\ t$ with Set and drop the symbols Ty and set and the rule $\mathit{Ty}\ \mathit{set} \hookrightarrow \mathit{Set}$. Then, we can drop the symbol $\mathit{Set1}$ as well.

So the only difference with the Calculus of constructions without $\triangle$ is that translations of contexts now contain variables of type Set, that translate the variables of type $\Box$.

5.12. **The Calculus of constructions with a constant $\iota : \Box$.** Adding the axiom (ι) to the Calculus of constructions yields a sub-theory with the 10 axioms (Set), (El), (ι), (Prop), (Prf), $(\Rightarrow_d)$, $(\forall)$, (o), $(\rightsquigarrow_d)$, and (π). It corresponds to the Calculus of constructions with an extra constant ι of type $\Box$. Adding a constant of type Set in $\lambda\Pi/\equiv$, like adding variables of type Set does not require to introduce an extra sort $\triangle$.

Some developments in the Calculus of constructions choose to declare the types of mathematical objects such as ι, *nat*, etc. in $*$, that would correspond to $\iota : \mathit{Prop}$, fully identifying types and propositions. The drawback of this choice is that it gives the type $*$ to the type ι of the constant 0, and the type $\Box$ to the type $\iota \to *$ of the constant *positive*, while, in Simple type theory, both ι and $\iota \to o$ are simple types. This is the reason why, in the theory $\mathcal{U}$, we give the type Set and not the type Prop to the constant ι. So, the expression of the simple type $\iota \to o$ uses the constant $\rightsquigarrow_d$, that is, type constructors, as both ι and o have type Set, and not the constant π, dependent types, that would be used if ι had the type Prop and o the type Set. Dependent types, the constant π, are thus marginalized to type functions mapping proofs to terms.

5.13. **The Minimal sub-theory.** Adding the axioms $(\Rightarrow)$ and $(\rightsquigarrow)$ yields a sub-theory with the 12 axioms (Set), (El), (ι), $(Prop)$, (Prf), $(\Rightarrow)$, $(\forall)$, (o), $(\rightsquigarrow)$, $(\rightsquigarrow_d)$, $(\Rightarrow_d)$, and (π) called the "Minimal sub-theory" of the theory $\mathcal{U}$. It contains both the 10 axioms of the Calculus of constructions and the 9 axioms of Minimal simple type theory. It is a formulation of the Calculus of constructions where dependent and non dependent arrows are distinguished. It is not a genuine extension of the Calculus of constructions as, each time we use a non dependent constant $\rightsquigarrow$ or $\Rightarrow$, we can use the dependent ones instead: a term of the form $t \rightsquigarrow u$ can always been replaced with the term $t \rightsquigarrow_d \lambda x : El\, t, u$, where the variable x does not occur in u, and similarly for the implication. Thus, any proof expressed in the Minimal sub-theory, in particular any proof expressed in Minimal simple type theory, can always be translated to the Calculus of constructions.

Conversely, a proof expressed in the Calculus of constructions can be expressed in this theory. In a proof, every symbol $\rightsquigarrow_d$ or $\Rightarrow_d$ that uses a dummy dependency can be replaced with a symbol $\rightsquigarrow$ or $\Rightarrow$. Every proof that does not use $\rightsquigarrow_d$, $\Rightarrow_d$ and π, can be expressed in Minimal simple type theory.

5.14. **The Calculus of constructions with dependent types at the object level.** In the Calculus of constructions with a constant ι of type $\Box$, there are no dependent types at the object level. We have types $\iota \to \iota$ and $\iota \to *$, thanks to the rule $\langle \Box, \Box, \Box \rangle$, but no type $\iota \to \Box$. We can introduce such dependent types by adding an extra sort $\triangle$, together with an axiom $\Box : \triangle$ and a rule $\langle \Box, \triangle, \triangle \rangle$. We obtain this way a Pure type system whose expression in $\lambda\Pi/\equiv$ [CD07] contains 13 declarations and 7 rules.

Using the same notations as above, $Prop$ for U_*, Set for $U_\Box$, $Set1$ for $U_\triangle$, etc., we get exactly the 13 axioms $(Prop)$, (Prf), (Set), (El), $(Set1)$, (Ty), (o), (set), $(\Rightarrow_d)$, (π), $(\forall)$, $(\rightsquigarrow_d)$, and $(\rightsquigarrow_d)$. The theory formed with these 13 axioms is thus equivalent to more common formulations of the Calculus of constructions with dependent types at that object level.

5.15. **The Calculus of constructions with prenex predicative polymorphism.** In the Calculus of constructions with an extra sort $\triangle$, polymorphism at the object level can be added with the rule $\langle \triangle, \Box, \Box \rangle$ that allows to build terms of the form $\Pi x : \Box, x \to x : \Box$ at the expense of making the system inconsistent [Hur95, Coq86]. Thus, just like in Simple type theory, we restrict to prenex predicative polymorphism: so, besides the sort $\triangle$, whose only closed irreducible element is $\Box$, we introduce a sort $\diamond$ for schemes and two rules $\langle \triangle, \Box, \diamond \rangle$ to build schemes by quantifying over an element of $\triangle$, that is, over $\Box$, in a type and $\langle \triangle, \diamond, \diamond \rangle$ to build schemes by quantifying over $\Box$ in another scheme. We also add a rule $\langle \triangle, *, * \rangle$ to quantify over $\Box$ in a proposition.

Alternatively, the Calculus of constructions with prenex predicative polymorphism can be defined as a cumulative type system [Bar99], making $\Box$ a subsort of $\diamond$ and having just one rule $\langle \triangle, \diamond, \diamond \rangle$ to quantify over a variable of type $\Box$ in a scheme and a rule $\langle \triangle, *, * \rangle$ to quantify over $\Box$ in a proposition. As there is no function whose co-domain is $\Box$, this subtyping does not need to propagate to product types.

Expressing this Cumulative type system in $\lambda\Pi/\equiv$ introduces 8 declarations and 4 rules on top of the Calculus of constructions: 2 declarations for the sort $\triangle$

- a constant $U_\triangle$ of type `TYPE`,
- and a constant $\varepsilon_\triangle$ of type $U_\triangle \to$ `TYPE`,

1 declaration and 1 rule for the axiom $\Box : \triangle$

- a constant $\square$ of type $U_\triangle$,
- and a rule $\varepsilon_\triangle\ \square \hookrightarrow \mathit{Set}$ (remember that Set is $U_\square$),

2 declarations for the sort $\diamond$

- a constant $U_\diamond$, that we write Scheme, of type $\mathtt{TYPE}$,
- and a constant $\varepsilon_\diamond$, that we write Els, of type $\mathit{Scheme} \to \mathtt{TYPE}$,

1 declaration and 1 rule to express that $\square$ is a subtype of $\diamond$

- a constant $\uparrow$ of type $\mathit{Set} \to \mathit{Scheme}$,
- and a rule $\mathit{Els}\ (\uparrow\ x) \hookrightarrow \mathit{El}\ x$ (remember that El is $\varepsilon_\square$),

1 declaration and 1 rule for the rule $\langle\triangle, \diamond, \diamond\rangle$

- a constant $\dot{\Pi}_{\langle\triangle,\diamond,\diamond\rangle}$, that we write $\forall$, of type $\Pi z : U_\triangle, (\varepsilon_\triangle\ z \to \mathit{Scheme}) \to \mathit{Scheme}$,
- and a rule $\mathit{Els}\ (\forall\ x\ y) \hookrightarrow \Pi z : \varepsilon_\triangle\ x, \mathit{Els}\ (y\ z)$,

and 1 declaration and 1 rule for the rule $\langle\triangle, *, *\rangle$

- $\dot{\Pi}_{\langle\triangle,*,*\rangle}$, that we write $\forall'$, of type $\Pi z : U_\triangle, (\varepsilon_\triangle\ z \to \mathit{Prop}) \to \mathit{Prop}$,
- and a rule $\mathit{Prf}\ (\forall'\ x\ y) \hookrightarrow \Pi z : \varepsilon_\triangle\ x, \mathit{Prf}\ (y\ z)$.

But, again, this theory can be simplified. Indeed, just like $\square$ is the only closed irreducible term of type $\triangle$, $\square$ is the only closed irreducible term of type $U_\triangle$ and thus for any closed term t of type $U_\triangle$, the term $\varepsilon_\triangle\ t$ reduces to Set. So in the type of the constants $\forall$ and $\forall'$: $\Pi z : U_\triangle, (\varepsilon_\triangle\ z \to \mathit{Scheme}) \to \mathit{Scheme}$ and $\Pi z : U_\triangle, (\varepsilon_\triangle\ z \to \mathit{Prop}) \to \mathit{Prop}$, we can replace the expression $\varepsilon_\triangle\ z$ with Set. Then, as there is no point in building a function space whose domain is a singleton, we can simplify these type further to $(\mathit{Set} \to \mathit{Scheme}) \to \mathit{Scheme}$ and $(\mathit{Set} \to \mathit{Prop}) \to \mathit{Prop}$. Accordingly, the associated reduction rules simplify to

$$\mathit{Els}\ (\forall\ y) \hookrightarrow \Pi z : \mathit{Set}, \mathit{Els}\ (y\ z)$$

and

$$\mathit{Prf}\ (\forall'\ y) \hookrightarrow \Pi z : \mathit{Set}, \mathit{Prf}\ (y\ z)$$

Then we can drop the symbols $\varepsilon_\triangle$ and $\square$, the rule $\varepsilon_\triangle\ \square \hookrightarrow \mathit{Set}$, and the symbol $U_\triangle$. We are left with the 5 declaration and 3 rules

- $\mathit{Scheme} : \mathtt{TYPE}$ (Scheme-decl)
- $\mathit{Els} : \mathit{Scheme} \to \mathtt{TYPE}$ (Els-decl)
- $\uparrow : \mathit{Set} \to \mathit{Scheme}$ ($\uparrow$-decl)
- $\mathit{Els}\ (\uparrow\ x) \hookrightarrow \mathit{El}\ x$ ($\uparrow$-red)
- $\forall : (\mathit{Set} \to \mathit{Scheme}) \to \mathit{Scheme}$ ($\forall$-decl)
- $\mathit{Els}\ (\forall\ y) \hookrightarrow \Pi z : \mathit{Set}, \mathit{Els}\ (y\ z)$ ($\forall$-red)
- $\forall' : (\mathit{Set} \to \mathit{Prop}) \to \mathit{Prop}$ ($\forall'$-decl)
- $\mathit{Prf}\ (\forall'\ y) \hookrightarrow \Pi z : \mathit{Set}, \mathit{Prf}\ (y\ z)$ ($\forall'$-red)

that is, the 5 axioms (Scheme), (Els), ($\uparrow$), ($\forall$), and ($\forall'$). Adding these 5 axioms to the 10 axioms defining the Calculus of constructions yields the 15 axioms (Set), (El), (ι), (Prop), (Prf), ($\Rightarrow_d$), ($\forall$), (o), ($\rightsquigarrow_d$), (π), (Scheme), (Els), ($\uparrow$), ($\forall$), and ($\forall'$) defining the Calculus of constructions with prenex predicative polymorphism [Thi20].

6. Conclusion

The theory $\mathcal{U}$ is thus a candidate for a universal theory where proofs developed in various proof systems: HOL Light, Isabelle/HOL, HOL 4, Coq, Matita, Lean, PVS, etc. can be expressed. This theory can be complemented with other axioms to handle inductive types, recursive functions, universes, etc. [Ass15, Thi20, Gen20]. Note however that the various axioms currently proposed for encoding recursive functions are based on rewriting and may be difficult to translate to systems requiring termination proofs. Using recursors should make this much easier.

Each proof expressed in the theory $\mathcal{U}$ can use a sub-theory of the theory $\mathcal{U}$, as if the other axioms did not exist: the classical connectives do not impact the constructive ones, propositions as objects and functionality do not impact predicate logic, dependent types and predicate subtyping do not impact simple types, etc.

The proofs in the theory $\mathcal{U}$ can be classified according to the axioms they use, independently of the system they have been developed in. Finally, some proofs using classical connectives and quantifiers, propositions as objects, functionality, dependent types, or predicate subtyping may be translated into smaller fragments and used in systems different from the ones they have been developed in, making the theory $\mathcal{U}$ a tool to improve the interoperability between proof systems.

In some cases, a proof can be directly transferred from one system to the other if it does not use some axioms. For instance, [Wan16] showed that many proofs coming from HOL were in fact constructive. However, we usually need to apply some transformations on proofs to transfer them from one sub-theory to the other. For instance, by replacing a dependent arrow by a non-dependent one when the second argument is not actually dependent, by applying some morphism on type universes [Thi20], or by trying to eliminate some uses of the excluded middle [Cau16], which is part of the axioms of Isabelle/HOL [Pau21], Lean [Car19] and automated theorem provers.

Some of the sub-theories of $\mathcal{U}$ are known to be consistent, but one may wonder whether the theory $\mathcal{U}$ itself is consistent. We conjecture that it is but leave this difficult problem for future work. A solution may be to extend the model developed by the second author in [Dow17] for proving the consistency of the encoding of HOL.

Acknowledgments. The authors want to thank Michael Färber, César Muñoz, Thiago Felicissimo, and Makarius Wenzel for helpful remarks on a first version of this paper, as well as the anonymous reviewers for their useful comments.

References

[ABC+16] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the lambda-Pi-calculus modulo theory. Manuscript, 2016.

[AH14] L. Allali and O. Hermant. Semantic A-translation and super-consistency entail classical cut elimination. *CoRR*, abs/1401.0998, 2014. `arXiv:1401.0998`.

[Ass15] A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, 2015.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science. Volume 2. Background: computational structures*, pages 117–309. Oxford University Press, 1992.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, France, 1999.

[BDG+21] F. Blanqui, G. Dowek, E. Grienenberger, G. Hondet, and F. Thiré. Some axioms for mathematics. In *Proceedings of the 6th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 195, 2021.

[Ber88] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Manuscript, 1988.

[BGH19] F. Blanqui, G. Genestier, and O. Hermant. Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting. In *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 131, 2019.

[BH21] F. Blanqui and G. Hondet. Encoding of predicate subtyping and proof irrelevance in the $\lambda\pi$-calculus modulo theory. In *Proceedings of the 26th International Conference on Types for Proofs and Programs*, Leibniz International Proceedings in Informatics 188, 2021.

[Bla01] F. Blanqui. *Type theory and rewriting*. PhD thesis, Université Paris-Sud, France, 2001.

[Bla20] F. Blanqui. Type Safety of Rewrite Rules in Dependent Types. In *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 167, 2020.

[Car19] M. Carneiro. The type theory of Lean. `https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf`, 2019.

[Cau16] R. Cauderlier. A rewrite system for proof constructivization. In *Proceedings of the 11th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, ACM International Conference Proceeding Series, 2016. `doi:10.1145/2966268.2966270`.

[CD07] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 4583, 2007.

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[Chu40] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[Coq86] Th. Coquand. An analysis of Girard's paradox. Technical Report RR-0531, Inria, 1986. URL: `https://hal.inria.fr/inria-00076023`.

[DHK03] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31:33–72, 2003.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. North-Holland, 1990.

[Dor10] A. Dorra. équivalence de curry-howard entre le $\lambda\Pi$ calcul et la logique intuitionniste. Internship report, 2010.

[Dow15] G. Dowek. On the definition of the classical connectives and quantifiers. In E.H. Haeusler, W. de Campos Sanz, and B. Lopes, editors, *Why is this a Proof?, Festschrift for Luiz Carlos Pereira*. College Publications, 2015.

[Dow17] G. Dowek. Models and Termination of Proof Reduction in the $\lambda\Pi$-Calculus Modulo Theory. In *Proceedings of the 44th International Colloquium on Automata, Languages and Programming*, Leibniz International Proceedings in Informatics 80, 2017.

[DW03] G. Dowek and B. Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68(4):1289–1316, 2003.

[DW05] G. Dowek and B. Werner. Arithmetic as a theory modulo. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2005.

[FT19] Gaspard Férey and François Thiré. Proof Irrelevance in LambdaPi Modulo Theory. `https://eutypes.cs.ru.nl/eutypes_pmwiki/uploads/Main/books-of-abstracts-TYPES2019.pdf`, 2019.

[Gen20] G. Genestier. *Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting*. PhD thesis, Université Paris-Saclay, 2020.

[Geu95] H. Geuvers. The Calculus of Constructions and Higher Order Logic. In Ph. de Groote, editor, *The Curry-Howard isomorphism*, volume 8 of *Cahiers du Centre de logique*, pages 139–191. Université catholique de Louvain, 1995.

[Gil18] F. Gilbert. *Extending higher-order logic with predicate subtyping: Application to PVS. (Extension de la logique d'ordre supérieur avec le sous-typage par prédicats)*. PhD thesis, Sorbonne Paris Cité, France, 2018.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris VII, 1972.

[Gri19] É. Grienenberger. A logical system for an Ecumenical formalization of mathematics, 2019. Manuscript.

[Gri21] É. Grienenberger. Expressing Ecumenical systems in the lambda-pi-calculus modulo theory, 2021. In preparation.

[HA28] D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Springer-Verlag, 1928.

[HB20] G. Hondet and F. Blanqui. The New Rewriting Engine of Dedukti. In *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 167, 2020.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

[Hur95] A. J. C. Hurkens. A simplification of Girard's paradox. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[KvOvR93] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[NM88] G. Nadathur and D. Miller. An overview of lambda-prolog. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 810–827, 1988.

[OS97] Sam Owre and Natarajan Shankar. *The Formal Semantics of PVS*. SRI International, SRI International, Computer Science Laboratory, Menlo Park CA 94025 USA, 1997.

[Pau93] L.C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993.

[Pau21] L. Paulson. Isabelle's logics, 2021.

[PR17] L.C. Pereira and R.O. Rodriguez. Normalization, soundness and completeness for the propositional fragment of Prawitz'Ecumenical system. *Revista Portuguesa de Filosofia*, 73(3-4):1153–1168, 2017.

[Pra15] D. Prawitz. Classical versus intuitionistic logic. In E.H. Haeusler, W. de Campos Sanz, and B. Lopes, editors, *Why is this a Proof?, Festschrift for Luiz Carlos Pereira*. College Publications, 2015.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[Ter89] J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Manuscript, 1989.

[TeR03] TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[Thi18] F. Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. In Frédéric Blanqui and Giselle Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, volume 274 of *EPTCS*, pages 57–71, 2018.

[Thi20] F. Thiré. *Interoperability between proof systems using the Dedukti logical framework*. PhD thesis, Université Paris-Saclay, France, 2020.

[Wan16] S. Wang. Higher order proof engineering: Proof collaboration, transformation, checking and retrieval. https://hal.inria.fr/hal-01250197/document, 2016. Presented at AITP'16.