Some Axioms for Mathematics

Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

Émilie Grienenberger
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

Gabriel Hondet
Université Paris-Saclay, ENS Paris-Saclay, LMF, CNRS, Inria, France

François Thiré
Nomadic Labs, Paris, France

Abstract

The $\lambda\Pi$-calculus modulo theory is a logical framework in which many logical 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.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Type theory; Theory of computation → Equational logic and rewriting

Keywords and phrases logical framework, axiomatic theory, dependent types, rewriting, interoperabilty

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.20

Acknowledgements The authors want to thank Michael Färber, César Muñoz, Thiago Felicissimo, and Makarius Wenzel for helpful remarks on a first version of this paper.

1 Introduction

The $\lambda\Pi$-calculus modulo theory ($\lambda\Pi/\equiv$) [13], implemented in the system Dedukti [3, 29], is a logical framework, that is a framework to define theories. It generalizes some previously proposed frameworks: Predicate logic [28], λ-Prolog [32], Isabelle [34], the Edinburgh logical framework [27], also called the $\lambda\Pi$-calculus, Deduction modulo theory [17, 18], Pure type systems [6, 39], and Ecumenical logic [36, 16, 35, 25]. 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.

$\lambda\Pi/\equiv$ enables to express all theories that can be expressed in Predicate logic, such as geometry, arithmetic, and set theory, but also Simple type theory [10] and the Calculus of constructions [12], that are less easy to define in Predicate logic.

We present a theory in $\lambda\Pi/\equiv$, 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,

© Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré;
licensed under Creative Commons License CC-BY 4.0

HOL Light, Isabelle/HOL, etc.), the Calculus of constructions (such as Coq, Matita, Lean, 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. 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 systems. Finally, in Section 5, we detail the sub-theories of $\mathcal{U}$ that correspond to the above mentioned systems.

2 The $\lambda\Pi$-calculus modulo theory

$\lambda\Pi/\equiv$ is an extension of the Edinburgh logical framework [27] with a primitive notion of computation defined with rewriting rules [14, 38].

The terms are those of the Edinburgh logical framework

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

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 *const*(t).

A rewriting rule is a pair of terms $\ell \hookrightarrow r$, such that $\ell = c\ \ell_1\ \dots\ \ell_n$, 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 $\hookrightarrow_{\beta\mathcal{R}}$.

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 $x_1 : B_1, \dots, x_m : B_m$ formed with a variable and a term. A signature Σ is a list of declarations $c_1 : A_1, \dots, c_n : A_n$ formed with a constant and a closed term, that is a term term with no free variables. This is why the rule (const) requires no context for typing A. We write $|\Sigma|$ for the set $\{c_1, \dots, c_n\}$, and $\Lambda(\Sigma)$ for the set of terms t such that *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 $x_1 : B_1, \dots, x_m : B_m$ is in $\Lambda(\Sigma)$ if $B_1, \dots, B_m$ 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 [7]) for, in this case, the product is injective modulo $\equiv_{\beta\mathcal{R}}$: $\Pi x : A, B \equiv_{\beta\mathcal{R}} \Pi x : A', B'$ iff $A \equiv_{\beta\mathcal{R}} A'$ 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 $\Gamma \vdash_{\Sigma,\mathcal{R}} \theta r : A$.

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

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

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

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

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

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

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

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

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

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

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 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 7:

▶ **Lemma 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 $\mathcal{U}$

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.

The easiest way to build the notion of object-term in $\lambda\Pi/\equiv$ would be to declare a constant I of type $\texttt{TYPE}$ and constants of type $I \to \ldots \to I \to I$ 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 $(succ\ x)$, would then just be $\lambda\Pi/\equiv$ terms of type I and, in an object-term, the variables would be $\lambda\Pi/\equiv$ 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. But these sorts would be mixed with the other terms of type TYPE, which we will introduce later. Instead, we declare a constant *Set* of type TYPE, a constant ι of type *Set*, and a constant *El* to embed the terms of type *Set* into terms of type TYPE

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

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

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

so that the symbol I can be replaced with the term $El\ \iota$. If we want to have object-terms of several sorts, we declare several constants ι_1, ι_2, etc. of type *Set*. The types of object-terms then have the form $El\ A$ and are distinguished among the other terms of type TYPE.

Assigning the type $Set \to \texttt{TYPE}$ to the constant *El* uses the fact that $\lambda\Pi/\equiv$ supports dependent types.

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 are then just constants of type $El\ \iota \to \ldots \to El\ \iota \to Prop$. Propositions are then $\lambda\Pi/\equiv$ terms of type *Prop*.

Implication

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

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

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. So we introduce an embedding *Prf* of propositions into types, mapping each proposition A to the type $Prf\ A$ of its proofs

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

Note that this embedding is not surjective. In particular *Set*, $El\ \iota$, and *Prop* are not types of proofs. So, there are more types than propositions, and propositions and types are not fully identified.

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. 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 $\mathit{Prf}\ A \to \mathit{Prf}\ A$

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

In the theory $\mathcal{U}$, the Brouwer-Heyting-Kolmogorov interpretation of proofs for implication is made explicit: it is the rule ($\Rightarrow$-red).

Universal quantification

Unlike implication, the universal quantifier binds a variable. Thus, we express the proposition $\forall z\ A$ as the proposition $\forall\ (\lambda z : \mathit{El}\ \iota, A)$ [10, 32, 34, 27], yielding the type $(\mathit{El}\ \iota \to \mathit{Prop}) \to \mathit{Prop}$ for the constant $\forall$ itself. But, we want to allow quantification over variables of any type $\mathit{El}\ B$, for B of type Set. Thus, we generalize this type to

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

and we write $\forall\ \iota\ (\lambda z : \mathit{El}\ \iota, A)$ 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 $\mathit{El}\ x$ to a proof of $p\ z$

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

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

Other constructive connectives and quantifiers

We define the other connectives and quantifiers, *à la* Russell, for instance $\mathit{Prf}\ (x \wedge y)$ as $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$. In this definition, we do not use the quantifier $\forall$ of the theory $\mathcal{U}$ (so far, in the theory $\mathcal{U}$, we can quantify over the type $\mathit{El}\ \iota$, 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 $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$ has type `TYPE` and not Prop. But, the rule rewriting $\mathit{Prf}\ (x \wedge y)$ to $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$ introduces some impredicativity, as $x \wedge y$ of type Prop is "defined" as the inverse image, for the embedding Prf, of the type $\Pi z : \mathit{Prop}, (\mathit{Prf}\ x \to \mathit{Prf}\ y \to \mathit{Prf}\ z) \to \mathit{Prf}\ z$, that contains a quantification on a variable of type Prop

$$\top : \mathit{Prop} \qquad (\top\text{-decl})$$
$$\mathit{Prf}\ \top \ \hookrightarrow\ \Pi z : \mathit{Prop}, \mathit{Prf}\ z \to \mathit{Prf}\ z \qquad (\top\text{-red})$$
$$\bot : \mathit{Prop} \qquad (\bot\text{-decl})$$
$$\mathit{Prf}\ \bot \ \hookrightarrow\ \Pi z : \mathit{Prop}, \mathit{Prf}\ z \qquad (\bot\text{-red})$$
$$\neg : \mathit{Prop} \to \mathit{Prop} \qquad (\neg\text{-decl})$$
$$\mathit{Prf}\ (\neg\, x) \ \hookrightarrow\ \mathit{Prf}\ x \to \Pi z : \mathit{Prop}, \mathit{Prf}\ z \qquad (\neg\text{-red})$$

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

Infinity

Now that we have the symbols $\top$ and $\bot$, we can express that the type $El\ \iota$ 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* [19]. This choice of notation enables the definition of natural numbers as some elements of type $El\ \iota$

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

Classical connectives and quantifiers

The disjunction in constructive logic and in classical logic are governed by different deduction rules, thus they have a different meaning, and 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. These constructive and classical disjunctions need not belong to different languages, but they can coexist in the same Ecumenical one [36, 16, 35, 25].

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. In the theory $\mathcal{U}$, we have chosen to define the classical connectives and quantifiers as in [1], for instance $A \vee_c B$ as $(\neg\neg A) \vee (\neg\neg B)$. Using these definitions, the proposition $(P \wedge_c Q) \Rightarrow_c P$ is $(\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P)$, which is not exactly the negative translation $\neg\neg((\neg\neg((\neg\neg P) \wedge (\neg\neg Q))) \Rightarrow (\neg\neg P))$ of $(P \wedge Q) \Rightarrow P$, as the double negation at the root of the proposition is missing. As we already have a distinction between the proposition A and the type $Prf\ A$ of its proofs, we can just include this double negation into the constant *Prf*, introducing a classical version Prf_c of this constant

$$Prf_c : Prop \to \mathtt{TYPE} \qquad (Prf_c\text{-decl})$$
$$Prf_c \ \hookrightarrow\ \lambda x : Prop, Prf\ (\neg\neg x) \qquad (Prf_c\text{-red})$$
$$\Rightarrow_c : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\Rightarrow_c\text{-decl})$$
$$\Rightarrow_c \ \hookrightarrow\ \lambda x : Prop, \lambda y : Prop, (\neg\neg x) \Rightarrow (\neg\neg y) \qquad (\Rightarrow_c\text{-red})$$
$$\wedge_c : Prop \to Prop \to Prop \quad \text{(written infix)} \qquad (\wedge_c\text{-decl})$$
$$\wedge_c \ \hookrightarrow\ \lambda x : Prop, \lambda y : Prop, (\neg\neg x) \wedge (\neg\neg y) \qquad (\wedge_c\text{-red})$$

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

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.

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 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, once we have reconstructed Predicate logic, we can either define Simple type theory as a theory in Predicate logic or as an extension of Predicate logic. In the theory $\mathcal{U}$, we choose the second option, which 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 $El\ \iota$. In particular, we cannot quantify over propositions. To do so, we just need to declare a constant o of type Set and a rule identifying $El\ o$ and $Prop$

$$o : Set \qquad (o\text{-decl})$$
$$El\ o \hookrightarrow Prop \qquad (o\text{-red})$$

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

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

Functionality

Besides ι and o, we introduce more types in the theory, for functions and sets. To do so, we declare a constant $\leadsto$ and a rewriting rule

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

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

Dependent function types

The axiom ($\leadsto$) 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

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

Note that, if we apply the constant $\leadsto_d$ to a term t and a term $\lambda z : El\ t, u$, where the variable z does not occur in u, then $El\ (t \leadsto_d \lambda z : El\ t, u)$ rewrites to $El\ t \to El\ u$, just like $El\ (t \leadsto u)$. Thus, the constant $\leadsto_d$ 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. Just like we have a constant ι of type Set, we could add a constant $array$ of type $El\ \iota \to Set$ such that $array\ n$ is the type of arrays of length n. We could then construct the term $(\iota \leadsto_d \lambda x : El\ \iota, array\ x)$ of type Set and the type $El\ (\iota \leadsto_d \lambda x : El\ \iota, array\ x)$ that rewrites to $\Pi x : El\ \iota, El\ (array\ x)$, would be the type of functions mapping a natural number n to an array of length n. So, this symbol $\leadsto_d$ becomes useful, only if we add such a constant $array$, object-level dependent types, or the symbols π or $psub$ below.

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

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

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. To be able to so, we introduce another constant π and the corresponding rewriting rule

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

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

$$El\ (\iota \leadsto \iota \leadsto_d \lambda y : El\ \iota, \pi\ (positive\ y)\ (\lambda z : Prf\ (positive\ y), \iota))$$

If we also have a constant eq_ι of type $El\ (\iota \leadsto \iota \leadsto o)$, we can then express the proposition

$$positive\ y \Rightarrow_d \lambda p : Prf\ (positive\ y), eq_\iota\ (div\ x\ y\ p)\ (div\ x\ y\ p)$$

usually written $y > 0 \Rightarrow x/y = x/y$. The proposition $x/y = x/y$ is well-formed, but it contains 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.

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, even if we had a reflexivity axiom for the aforementioned equality eq_ι, the proposition

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

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 $\mathit{div}\ 1\ 0\ r$ [20, 9].

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 $\mathit{pair}\ \iota\ \mathit{positive}\ u\ p$ is written $\mathit{psub}\ \iota\ \mathit{positive}$, or informally $\{x : \iota \mid \mathit{positive}\ x\}$. It can be called "the type of positive numbers". It is a subtype of the type of natural numbers defined with the predicate $\mathit{positive}$. 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 [20, 9] 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

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

Prenex predicative type quantification in types

Using the symbols of the theory $\mathcal{U}$ introduced so far, the symbol for equality of elements of type ι is eq_ι of type $El\ (\iota \leadsto \iota \leadsto o)$. This equality symbol is not polymorphic. Indeed, it cannot be used to express the equality of, for example, functions of type $\iota \leadsto \iota$. This motivates the introduction of object-level polymorphism [24, 37]. However extending Simple type theory with object-level polymorphism makes it inconsistent [30, 11], 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, a constant *Els* to embed the terms of type *Scheme* into terms of type TYPE, a constant $\uparrow$ to embed the terms of type *Set* into terms of type *Scheme* and a rule connecting these embeddings

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

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

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

This way, we can give the polymorphic type $Els\ (\forall\ (\lambda A : Set, \uparrow (A \leadsto A \leadsto o)))$ to the equality *eq*. In the same way, the type of the identity function is $Els\ (\forall\ (\lambda A : Set, \uparrow (A \leadsto A)))$. It rewrites to $\Pi A : Set, El\ A \to El\ A$. Therefore, it is inhabited by the term $\lambda A : Set, \lambda x : El\ A, x$.

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

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

This way, the reflexivity of equality can be expressed as $(\forall\ (\lambda A : Set, \forall\ A\ (\lambda x : El\ A, eq\ A\ x\ x)))$.

The theory $\mathcal{U}$: bringing everything together

The theory $\mathcal{U}$ is formed with the 38 axioms with a black bar at the beginning of the line: (*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), ($\leadsto$), ($\leadsto_d$), ($\Rightarrow_d$), (π), (0), (*succ*), (*pred*), (*positive*), (*psub*), (*pair*), ($pair^{\dagger}$), (*fst*), (*snd*), (*Scheme*), (*Els*), ($\uparrow$), ($\forall$), ($\forall$). Note that, strictly speaking, the declaration ($pair^{\dagger}$-decl) and the rule (*pair*-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, 12 only have a constant declaration, 24 have a constant declaration and one rewriting rule, and 2 have a constant declaration and two rewriting rules. So $\Sigma_{\mathcal{U}}$ contains 38 declarations and $\mathcal{R}_{\mathcal{U}}$ 28 rules.

This large number of axioms is explained by the fact that $\lambda\Pi/\equiv$ is a weaker framework than Predicate logic. The 19 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 19 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, for instance 4 axioms: (0), (*succ*), (*pred*), and (*positive*) are needed to express "the" axiom of infinity, 5 (*psub*), (*pair*), ($pair^\dagger$), (*fst*), and (*snd*) to express predicate subtyping, and 5 (*Scheme*), (*Els*), ($\uparrow$), ($\forall$), and ($\not\forall$) to express prenex polymorphism. 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 *positive* (*succ* x) rewrites to $\top$, if we want (*positive*) and (*succ*) in $\Sigma_{\mathcal{S}}$, we must include $\top$ in $\Sigma_{\mathcal{S}}$ and the rule rewriting *positive* (*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 3** (Fragment). *A signature Σ_1 is included in a signature Σ_0, $\Sigma_1 \subseteq \Sigma_0$, 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* $(c : A) \in \Sigma_1$, $const(A) \subseteq |\Sigma_1|$*;*
- *for all* $\ell \hookrightarrow r \in \mathcal{R}_0$, *if* $const(\ell) \subseteq |\Sigma_1|$, *then* $const(r) \subseteq |\Sigma_1|$ *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** (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 following 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 5** (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, $w \in \Lambda(\Sigma_1)$, $u \hookrightarrow_1^* w$ and $v \hookrightarrow_1^* w$. ◀

▶ **Definition 6** (Sub-theory)**.** *A system $\Sigma_1, \mathcal{R}_1$ is a sub-theory of a theory $\Sigma_0, \mathcal{R}_0$, if $\Sigma_1, \mathcal{R}_1$ 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 7.** *Let $\Sigma_0, \mathcal{R}_0$ be a confluent system and $\Sigma_1, \mathcal{R}_1$ be a fragment of $\Sigma_0, \mathcal{R}_0$ that preserves typing. 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.*

Proof. By induction on the derivation. The important 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

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

  as Γ, A, and t are in $\Lambda(\Sigma_1)$, by induction hypothesis, there exists A' in $\Lambda(\Sigma_1)$ such that $\texttt{TYPE} \hookrightarrow_0^* A'$ 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 $\texttt{TYPE}$ is a sort, $A' = \texttt{TYPE}$. Therefore, $\Gamma \vdash_1 A : \texttt{TYPE}$ is derivable.
  As B is typable and every subterm of a typable term is typable, $\texttt{KIND}$ does not occur in B. As $B \hookrightarrow_0^* B'$ and no rule contains $\texttt{KIND}$, $\texttt{KIND}$ does not occur in B' as well. Hence, $B' \neq \texttt{KIND}$. By Lemma 2, as $\Gamma, x : A \vdash_1 t : B'$ is derivable and $B' \neq \texttt{KIND}$, 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, 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, as $\Gamma \vdash_1 t : \Pi x : A_1, B_1$ is derivable and $\Pi x : A_1, B_1 \neq \texttt{KIND}$, there exists a sort s such that $\Gamma \vdash_1 \Pi x : A_1, B_1 : s$ is derivable. Thus, by Lemma 2, $\Gamma \vdash_1 A_1 : \texttt{TYPE}$ 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

$$\frac{\Gamma \vdash_0 t : A \quad \Gamma \vdash_0 B : s}{\Gamma \vdash_0 t : B}\ (\text{conv}) \quad A \equiv_{\beta\mathcal{R}_0} B$$

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, $C \in \Lambda(\Sigma_1)$ and $A' \hookrightarrow_1^* C$.
As B is typable and every subterm of a typable term is typable, `KIND` does not occur in B. As $B \hookrightarrow_0^* C$ and no rule contains `KIND`, `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, 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 $B \hookrightarrow_0^* D'$. ◀

▶ **Corollary 8.** *Let $\Sigma_0, \mathcal{R}_0$ be a confluent system, $\Sigma_1, \mathcal{R}_1$ be a fragment of $\Sigma_0, \mathcal{R}_0$ that preserves typing. 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, $\Sigma_1, \mathcal{R}_1$ be 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 we have $D \hookrightarrow_1^* D'$, and we conclude with the rule (conv). ◀

▶ **Theorem 9** (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 [31]. 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 [8, 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), $pred\ (succ\ x) \hookrightarrow x$, the left-hand side has type $El\ \iota$ if the equation $type(x) = El\ \iota$ is satisfied. Modulo this equation, the right-hand side has type $El\ \iota$ in any fragment containing the symbols of the rule.

In (fst-red), $fst\ t\ p\ (pair^\dagger\ t'\ p'\ m) \hookrightarrow m$, the left-hand side has type $El\ t$ if $type(t) = Set$, $type(p) = El\ t \to Prop$, $El\ (psub\ t'\ p') = El\ (psub\ t\ p)$, $type(t') = Set$, $type(p') = El\ t' \to Prop$, and $type(m) = El\ t'$. But, in $\mathcal{U}$, there is no rule of the form $El\ (psub\ t\ p) \hookrightarrow r$. Hence,

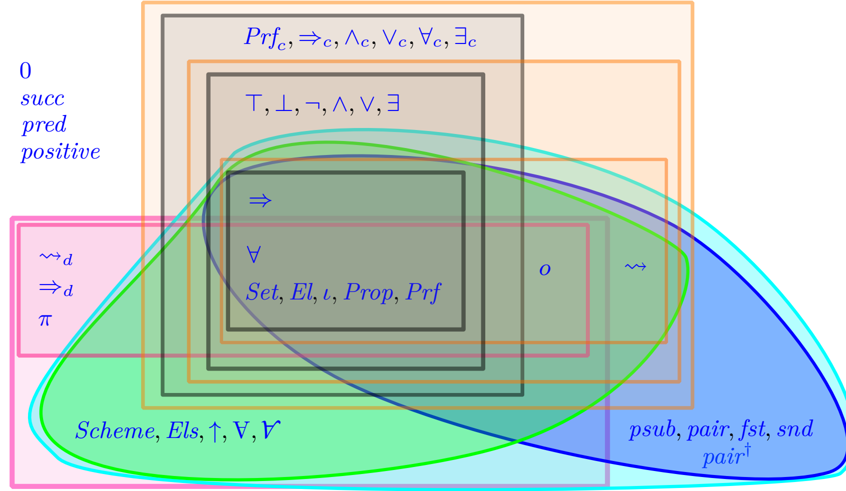

Figure 2 The wind rose. In black: Minimal, Constructive, and Ecumenical predicate logic. In orange: Minimal, Constructive, and Ecumenical simple type theory. In green: Simple type theory with prenex polymorphism. In blue: Simple type theory with predicate subtyping. In cyan: Simple type theory with predicate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

by confluence, the equation $El\ (psub\ t'\ p') = El\ (psub\ t\ p)$ is equivalent to the equations $t' = t$ and $p' = p$. Therefore, the right-hand side is of type $El\ t$ 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 13 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 8, if Γ, t, and A are in $\Lambda(\Sigma_\mathcal{S})$, and $\Gamma \vdash_{\Sigma_\mathcal{U}, \mathcal{R}_\mathcal{U}} t : A$, then $\Gamma \vdash_{\mathcal{R}_\mathcal{S}, \Sigma_\mathcal{S}} t : A$.

Minimal predicate logic. The 7 axioms (Set), (El), (ι), ($Prop$), (Prf), ($\Rightarrow$), and ($\forall$) define Minimal predicate logic. This theory can be proven equivalent to more common formulations of Minimal predicate logic. As Minimal predicate logic is itself a logical framework, it must be complemented with more axioms, such as the axioms of geometry, arithmetic, etc.

Constructive predicate logic. The 13 axioms (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 [15, 3].

Ecumenical predicate logic. The 19 axioms (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 [26]. 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.

Minimal simple type theory. The 9 axioms (*Set*), (ι), (*El*), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), (o), and ($\leadsto$) define Minimal simple type theory. And this theory can be proven equivalent to more common formulations of Minimal simple type theory [2, 3]. We could save the declaration (*Prop*-decl) and the rule (o-red) by replacing everywhere *Prop* with *El* o[3]. However, by removing (*Prop*-decl) and (o-red), this theory does not construct Simple type theory as an extension of Minimal predicate logic.

Constructive simple type theory. The 15 axioms (*Set*), (*El*), (ι), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), ($\top$), ($\bot$), ($\neg$), ($\wedge$), ($\vee$), ($\exists$), (o) and ($\leadsto$) define Constructive simple type theory.

Ecumenical simple type theory. The 21 axioms (*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 ($\leadsto$) define Ecumenical simple type theory. And this theory can be proven equivalent to more common formulations of Ecumenical simple type theory [26].

Simple type theory with predicate subtyping. Adding to the 9 axioms of Minimal simple type theory the 5 axioms of predicate subtyping yields Minimal simple type theory with predicate subtyping, formed with the 14 axioms (*Set*), (ι), (*El*), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), (o), ($\leadsto$), (*psub*), (*pair*), ($pair^\dagger$), (*fst*), and (*snd*). This theory can be proven equivalent to more common formulations of Minimal simple type theory with predicate subtyping [23, 9]. Such formulations like PVS [33] often use predicate subtyping implicitly to provide a lighter syntax without (*pair*), ($pair^\dagger$), (*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 Page 9 extends the conversion in order to ignore these proofs.

Simple type theory with prenex predicative polymorphism. Adding to Minimal simple type theory the 5 axioms of prenex predicative polymorphism yields Simple type theory with prenex predicative polymorphism (STT$\forall$) [40, 41] formed with the 14 axioms (*Set*), (*El*), (ι), (*Prop*), (*Prf*), ($\Rightarrow$), ($\forall$), (o), ($\leadsto$), (*Scheme*), (*Els*), ($\uparrow$), ($\forall\!\!\forall$), and ($\dot{\forall\!\!\forall}$).

Simple type theory with predicate subtyping and prenex polymorphism. Adding to the 9 axioms of Simple type theory both the 5 axioms of predicate subtyping and the 5 axioms of prenex polymorphism yields a sub-theory with 19 axioms which is a subsystem of PVS [33] handling both predicate subtyping and prenex polymorphism.

The Calculus of constructions. The 9 axioms (*Set*), (*El*), (*Prop*), (*Prf*), ($\Rightarrow_d$), ($\forall$), (o), ($\leadsto_d$), and (π) define the Calculus of constructions. This is the usual expression of the Calculus of constructions in $\lambda\Pi/\equiv$ [13, 3] except that we write *Prop* for U_*, *Prf* for ε_*, *Set* for $U_\Box$, *El* for $\varepsilon_\Box$, o for $\dot{*}$, $\Rightarrow_d$ for $\dot{\Pi}_{\langle *,*,* \rangle}$, $\forall$ for $\dot{\Pi}_{\langle \Box,*,* \rangle}$, π for $\dot{\Pi}_{\langle *,\Box,\Box \rangle}$, and $\leadsto_d$ for $\dot{\Pi}_{\langle \Box,\Box,\Box \rangle}$. As $\Rightarrow_d$ is $\dot{\Pi}_{\langle *,*,* \rangle}$, $\forall$ is $\dot{\Pi}_{\langle \Box,*,* \rangle}$, π is $\dot{\Pi}_{\langle *,\Box,\Box \rangle}$, and $\leadsto_d$ is $\dot{\Pi}_{\langle \Box,\Box,\Box \rangle}$, using the terminology of Barendregt's λ-cube [4], the axiom ($\forall$) expresses polymorphism, the axiom (π) dependent types, and the axiom ($\leadsto_d$) type constructors. Note that these constants have similar types.

So if Γ is a context and A is a term A in the Calculus of constructions then A is inhabited in Γ in the Calculus of constructions if and only if the translation of A in $\lambda\Pi/\equiv$ is inhabited in the translation of Γ in $\lambda\Pi/\equiv$ [13, 3]. In the translation of Γ in $\lambda\Pi/\equiv$, variables have a $\lambda\Pi/\equiv$ type of the form *Prf* u or *El* u, and none of them can have the type *Set*. But, in $\lambda\Pi/\equiv$, nothing prevents from declaring a variable of type *Set*. So, the formulation of

the Calculus of constructions in $\lambda\Pi/\equiv$ is in fact a conservative extension of the original formulation of the Calculus of constructions, where the judgement $x : \mathit{Set} \vdash x : \mathit{Set}$ can be derived. Allowing the declaration of variables of type Set in the Calculus of constructions usually requires to add a sort $\triangle$ and an axiom $\Box : \triangle$ [22]. This is not needed here.

The Calculus of constructions with a type ι. 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), $(\leadsto_d)$, 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. We did not make this choice in the theory $\mathcal{U}$, because, then, the type ι of the constant 0 has type $*$ and the type $\iota \to *$ of the constant $\mathit{positive}$ has type $\Box$, while, in Simple type theory, both ι and $\iota \to o$ are simple types. So the expression of the simple type $\iota \to o$ requires type constructors and not dependent types. Dependent types, the constant π, are thus marginalized to type functions mapping proofs to terms.

In the Calculus of constructions with a constant ι of type $\Box$, there are no dependent types and no polymorphism at the object level, the latter leading to an inconsistent system [30, 11]. There are no object-level dependent types in the theory $\mathcal{U}$, that is the type $\mathit{El}\ \iota \to \mathit{Set}$ of the symbol array is not equivalent to a term of the form $\varepsilon_\triangle\ A$, but such dependent types could be added. Polymorphism is discussed below.

The Minimal sub-theory. Adding the axioms $(\Rightarrow)$ and $(\leadsto)$ yields a sub-theory with the 12 axioms (Set), (El), (ι), (Prop), (Prf), $(\Rightarrow)$, $(\forall)$, (o), $(\leadsto)$, $(\leadsto_d)$, $(\Rightarrow_d)$, 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. A proof expressed in the Calculus of constructions can be expressed in this theory. In a proof, every symbol $\leadsto_d$ or $\Rightarrow_d$ that uses a dummy dependency can be replaced with a symbol $\leadsto$ or $\Rightarrow$. Every proof that does not use $\leadsto_d$, $\Rightarrow_d$ and π, can be expressed in Minimal simple type theory.

The Calculus of constructions with prenex predicative polymorphism. Adding the 5 axioms of prenex predicative polymorphism to the 10 axioms of the Calculus of constructions with a constant ι yields a sub-theory formed with the 15 axioms (Set), (El), (ι), (Prop), (Prf), $(\Rightarrow_d)$, $(\forall)$, (o), $(\leadsto_d)$, (π), (Scheme), (Els), $(\uparrow)$, $(\forall)$, and $(\dot{\forall})$ defining the Calculus of constructions with prenex predicative polymorphism. It is a cumulative type system [5], containing four sorts $*$, $\Box$, $\triangle$ and $\diamond$, with $* : \Box$, $\Box : \triangle$, and $\Box \preceq \diamond$, and besides the rules $\langle *, *, * \rangle$, $\langle *, \Box, \Box \rangle$, $\langle \Box, *, * \rangle$, $\langle \Box, \Box, \Box \rangle$, a 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 [41].

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, co-inductive types, universes, etc. [2, 41, 21].

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.

References

1. L. Allali and O. Hermant. Semantic A-translation and super-consistency entail classical cut elimination. *CoRR*, abs/1401.0998, 2014. arXiv:1401.0998.
2. A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, 2015.
3. 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.
4. 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, pages 117–309. Oxford University Press, 1992.
5. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, France, 1999.
6. S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Manuscript, 1988.
7. F. Blanqui. *Type theory and rewriting*. PhD thesis, Université Paris-Sud, France, 2001. URL: http://hal.inria.fr/inria-00105525.
8. 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. doi:10.4230/LIPIcs.FSCD.2020.13.
9. 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.
10. A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
11. Th. Coquand. An analysis of Girard's paradox. Technical Report RR-0531, Inria, 1986. URL: https://hal.inria.fr/inria-00076023.
12. Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.
13. 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. doi:10.1007/978-3-540-73228-0_9.
14. 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.
15. A. Dorra. équivalence de curry-howard entre le $\lambda\Pi$ calcul et la logique intuitionniste. Internship report, 2010.
16. 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.

17 G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31:33–72, 2003. doi:10.1023/A:1027357912519.

18 G. Dowek and B. Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68(4):1289–1316, 2003. doi:10.2178/jsl/1067620188.

19 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.

20 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.

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

22 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.

23 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.

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

25 É. Grienenberger. A logical system for an Ecumenical formalization of mathematics, 2019. Manuscript.

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

27 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.

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

29 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. doi:10.4230/LIPIcs.FSCD.2020.35.

30 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.

31 J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993. doi:10.1016/0304-3975(93)90091-7.

32 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.

33 Sam Owre and Natarajan Shankar. *The Formal Semantics of PVS.* SRI International, SRI International, Computer Science Laboratory, Menlo Park CA 94025 USA, 1997. URL: http://pvs.csl.sri.com/doc/semantics.pdf.

34 L.C. Paulson. Isabelle: The next 700 theorem provers. *CoRR*, cs.LO/9301106, 1993. arXiv:cs.LO/9301106.

35 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.

36 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.

37 J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.
38 TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
39 J. Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Manuscript, 1989.
40 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.
41 F. Thiré. *Interoperability between proof systems using the Dedukti logical framework*. PhD thesis, Université Paris-Saclay, France, 2020.