Reconciling Impredicative Axiom and Universe

Stefan Monnier

Reconciling Impredicative Axiom and Universe

Stefan Monnier

Université de Montréal - DIRO, Montréal, Canada

Impredicativity and type theory share a long history since Russel introduced the notion of types specifically to try and rule out the logical inconsistency that can be derived from arbitrary impredicative quantification. In modern type theory, impredicativity is most commonly (re)introduced in one of two ways : the traditional way found in systems like the Calculus of Constructions, Lean, Coq, and System F, is to make the bottom universe impredicative, typically called Prop ; the other way, used in HoTT [15], is to provide a propositional resizing axiom that makes it possible to move proof-irrelevant types to a lower universe.

Coq's Prop and HoTT's propositional resizing axiom both restrict the use of impredicative quantification to the definition of proof-irrelevant propositions, which suggests they may be closely related, yet the actual mechanism by which they allow it is very different, making it unclear how they compare to each other in terms of expressiveness and interactions with other axioms.

In this article we try to provide an answer to this question by axiomatizing a Prop universe. More specifically, we describe a predicative type theory augmented with a set of axioms inspired by HoTT's propositional resizing axiom and then show a quasi-equivalence between this system and a similar type theory augmented with a Prop universe. We start with a small PTS and then grow the system to UTT-style inductive types.

In practical terms, the syntactic nature of the proof means that such an approach can be used also to translate code between systems. Furthermore, this final result suggests that the kind of impredicativity provided by Coq's Prop universe is closely related to that offered by HoTT's propositional resizing.

1 Introduction

Russell introduced the notion of *type* and *predicativity* as a way to stratify definitions so as to prevent the logical inconsistencies exposed typically via some kind of diagonalization argument [7]. This stratification seems sufficient to protect us from such paradoxes, but it does not seem to be absolutely necessary either : systems such as System F are not predicative yet they are generally believed to be consistent. Impredicativity is not indispensable for a logic to be useful, and indeed systems like Agda [4] demonstrate that you can go a long way without it, yet many popular systems, like Coq [6], do include some limited form of impredicativity either for convenience or because of the proof-theoretic strength it provides.

While classical set theory introduces forms of impredicativity via axioms (such as the powerset axiom), in the context of type theory, until recently impredicativity was always introduced by allowing elements of a specific universe (traditionally called Prop) to be quantified over elements from a higher universe, as is done in System F and the Calculus

of Constructions [5]. More recently, Voevodsky [16] proposed to introduce impredicativity via the use of *resizing rules* which allow moving those types which obey some particular property to a smaller universe. The most common of those rules is the *propositional resizing axiom* used in Homotopy type theory [15].

The propositional resizing axiom states that any proposition that is proof-irrelevant, i.e. any type which can have at most one inhabitant, can be considered as living in the smallest universe. While the impredicativity of System F and the original Calculus of Constructions is not associated to any kind of proof irrelevance, virtually all proof assistants based on impredicative type theories restrict their Prop universe to be proof-irrelevant. Intuitively, the two approaches seem thus related since in both cases they restrict the use of impredicativity to propositions that are proof-irrelevant. Yet the mechanisms by which they are defined are very different, making it unclear how they compare to each other in terms of expressiveness and interactions with other axioms.

In this article, we attempt to show more precisely how they compare by proving a quasi-equivalence between a calculus using a Prop universe and one using *a* resizing axiom.

Our contributions are :

— A proof we call "quasi-equivalence" between uCCω, an impredicative pure type system with a tower of universes, and rCCω, its sibling based on the predicative subset pCCω extended with an axiom loosely inspired by HoTT's propositional resizing axiom and propositional truncation. Formally we show that we can encode uCCω in rCCω and that we can encode rCCω in u^{+}CCω, which is almost like uCCω but with an extra rule.

— An extension of that quasi-equivalence to calculi with inductive types uCCωI and rCCωI. The complexity of this extension depends on the details of how inductive types are introduced : we limit our inductive types to the higher universes, as was done in UTT [8].

The proofs of encodings take the form of syntactic rewrites from one system to another, in the tradition of syntactic models [3], and can thus open the door to the translation of definitions between such systems.

The rest of the paper is structured as follows : in Section 2 we show the syntax and typing rules of the systems which we will be manipulating ; in Section 3 we show a naive encoding exposing our general approach, and we show how it fails to deliver a proof of equivalence ; in Section 4 we present the actual rCCω and the corresponding encoding which shows it to be quasi-equivalent to uCCω ; in Section 5 we show how to extend this result to inductive types ; in Section 6 we discuss the limitations of our proof as well as the differences between our calculi and the existing systems they are meant to model ; we then conclude in Section 7.

2 Background

The calculi we use in this paper are all extensions of pure type systems (PTS) [1]. The base syntax of the terms is defined as follows :

$$\begin{array}{llll} \textit{(var)} & x, y, f, t & \in & \mathcal{V} \\ \textit{(sort)} & s & \in & \mathcal{S} \\ \textit{(term)} & e, \tau & ::= & s \mid x \mid (x : \tau_1) \to \tau_2 \mid \lambda x : \tau . e \mid e_1\ e_2 \end{array}$$

Terms can be either a sort s ; or a variable x ; or a function $\lambda x : \tau . e$ where x is the formal argument, τ is its type, and e is the body ; or an application $e_1\ e_2$ which calls the function e_1 with argument e_2 ; or the type $(x : \tau_1) \to \tau_2$ of a function where τ_1 is the type of the argument and τ_2 is the type of the result and where x is bound within τ_2. We can write $\tau_1 \to \tau_2$ if x does not occur in τ_2. A specific PTS is then defined by providing the tuple

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau \ : \ s}{\vdash \Gamma, x : \tau} \qquad \frac{\vdash \Gamma \qquad \Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau} \qquad \frac{\vdash \Gamma \qquad (s_1 : s_2) \in \mathcal{A}}{\Gamma \vdash s_1 \ : \ s_2}$$

$$\frac{\Gamma \vdash \tau_1 \ : \ s_1 \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 \ : \ s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (x{:}\tau_1) \to \tau_2 \ : \ s_3}$$

$$\frac{\Gamma \vdash e_1 \ : \ (x{:}\tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1 \ e_2 \ : \ \tau_2\{e_2/x\}} \qquad \frac{\Gamma \vdash (x{:}\tau_1) \to \tau_2 \ : \ s \qquad \Gamma, x{:}\tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e \ : \ (x{:}\tau_1) \to \tau_2}$$

$$\frac{\Gamma \vdash e \ : \ \tau_1 \qquad \Gamma \vdash \tau_1 \ \simeq \ \tau_2 \ : \ s}{\Gamma \vdash e \ : \ \tau_2} \qquad \frac{\Gamma \vdash (\lambda x{:}\tau_1.e_1) \ e_2 \ : \ \tau_2}{\Gamma \vdash (\lambda x{:}\tau.e_1) \ e_2 \ \simeq \ e_1\{e_2/x\} \ : \ \tau_2} \ (\beta)$$

Figure 1. Main typing rules of our PTS

$$\begin{array}{rcll} \mathcal{S} & = & \{\ \mathcal{U}_\ell & \mid\ \ell \in \mathbb{N}\ \} \\ \mathcal{A} & = & \{\ (\mathcal{U}_\ell : \mathcal{U}_{\ell+1}) & \mid\ \ell \in \mathbb{N}\ \} \\ \mathcal{R} & = & \{\ (\mathcal{U}_{\ell_1}, \mathcal{U}_{\ell_2}, \mathcal{U}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ \ell_1, \ell_2 \in \mathbb{N}\ \} \end{array}$$

Figure 2. Definition of pCCω as a PTS.

$(\mathcal{S}, \mathcal{A}, \mathcal{R})$ which defines respectively the set $\mathcal{S}$ of sorts, the axioms $\mathcal{A}$ that relate the various sorts, and the rules $\mathcal{R}$ specifying which forms of quantifications are allowed in this system.

Figure 1 shows the main typing rules of our PTS, where $\Gamma \vdash e \ : \ \tau$ is the main judgment saying that expression e has type τ in context Γ. We have two auxiliary judgments : $\vdash \Gamma$ says that Γ is a well-formed context, and the convertibility judgment $\Gamma \vdash e_1 \ \simeq \ e_2 \ : \ \tau$ says that e_1 and e_2 are convertible at type τ in context Γ. These are all standard rules. We do not present the congruence, reflexivity, and symmetry rules for the convertibility in the interest of space.

Here is an example of a simple PTS which defines the familiar System F :

$$\begin{array}{rcl} \mathcal{S} & = & \{\ *,\ \Box\ \} \\ \mathcal{A} & = & \{\ (* : \Box)\ \} \\ \mathcal{R} & = & \{\ (*, *, *),\ (\Box, *, *)\ \} \end{array}$$

Where $*$ is the universe of values and $\Box$ is the universe of types and the axiom $(* : \Box)$ expresses the fact that types classify values. The rule $(*, *, *)$ corresponds to the traditional "small λ" and says that functions can quantify over "values" (i.e. elements of the universe $*$) and return values and that such functions are themselves values, while the rule $(\Box, *, *)$ corresponds to the traditional "big Λ" and says that functions can also quantify over "types" (i.e. elements of the universe $\Box$) and return values, and that those functions are also values.

Figure 2 shows the definition of our base, predicative, calculus, we call pCCω. It is a very simple pure type system with a tower of universes. All the sorts have the form $\mathcal{U}_\ell$ where ℓ is called the universe level and $\mathcal{U}_0$ is the bottom universe. To keep things simple, our universes are not cumulative. We have not yet investigated the impact of cumulativity on our work.

2.1 Impredicativity

Informally, a definition is impredicative if it is quantified over a type which includes the definition itself. For example in System F the polymorphic identity function $id = \Lambda t.\lambda x{:}t.x$

$$\begin{array}{rcll} \mathcal{S} & = & \{\, \mathcal{U}_\ell & \mid\ \ell \in \mathbb{N} \,\} \\ \mathcal{A} & = & \{\, (\mathcal{U}_\ell : \mathcal{U}_{\ell+1}) & \mid\ \ell \in \mathbb{N} \,\} \\ \mathcal{R} & = & \{\, (\mathcal{U}_{\ell_1}, \mathcal{U}_{\ell_2}, \mathcal{U}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ \ell_1 \in \mathbb{N}, \ell_2 \in \mathbb{N}^* \,\} \\ & & \cup\ \{\, (\mathcal{U}_\ell, \mathcal{U}_0, \mathcal{U}_0) & \mid\ \ell \in \mathbb{N} \,\} \end{array}$$

Figure 3. Definition of uCCω as a PTS.

is quantified over any type t, including the type $\forall t.t \to t$ of *id* itself. This opens the door to self-application, e.g. $id[\forall t.t \to t]id$, which is a crucial ingredient in most logical paradoxes, although in the case of System F the impredicativity is tame enough that it is not possible to encode those paradoxes.

In System F, the rule $(\Box, *, *)$ is the source of impredicativity because it allows the creation of a function in $*$ which quantifies over elements that belong to the larger universe $\Box$ and which can hence include its own type. To make it predicative, we would need to use $(\Box, *, \Box)$ meaning that a function that quantifies over types and returns values would now belong to the universe $\Box$. This would prevent instantiating a polymorphic function with a type which is itself polymorphic, and would thus disallow $id[\forall t.t \to t]id$ although you would still be able to do $id[\mathsf{Int} \to \mathsf{Int}](id[\mathsf{Int}])$.

In contrast to System F, we can see that pCCω is predicative because its rules have the form $(\mathcal{U}_{\ell_1}, \mathcal{U}_{\ell_2}, \mathcal{U}_{\mathsf{max}(\ell_1,\ell_2)})$, thus ensuring that a function is always placed in a universe at least as high as the objects over which it quantifies.

The traditional way to add impredicativity to a system like pCCω is by adding rules of the form $(\mathcal{U}_\ell, \mathcal{U}_0, \mathcal{U}_0)$ which allow impredicative quantifications in the *bottom* universe $\mathcal{U}_0$. Such an impredicative bottom universe is traditionally called Prop.

2.2 Propositional resizing

In homotopy type theory (HoTT, [15]), instead of providing an impredicative universe, impredicativity is provided via an axiom called *propositional resizing*. This axiom applies to all types that are so-called *mere propositions*, which means that they satisfy the predicate *isProp* which states that this type is proof-irrelevant, and which can be defined as follows :

$$\mathit{isProp}\ \tau : (x{:}\tau) \to (y{:}\tau) \to x = y$$

The resizing axiom says that any type which is a mere proposition in a universe $\mathcal{U}_{\ell+1}$ can be "resized" to an equivalent one in the smaller universe $\mathcal{U}_\ell$. By repeated application, it follows that any mere proposition can be resized to belong to the bottom universe $\mathcal{U}_0$.

Accompanying this axiom, HoTT also provides a *propositional truncation* operation $||\cdot||$ which basically throws away the information content of a type, turning it into a mere proposition. It comes with the introduction form $|\cdot|$ such that if $e : \tau$, then $|e| : ||\tau||$ and with an elimination principle (let us call it $elim_{||}$) which says that if $|e_1| : ||\tau_1||$ and $e_2 : \tau_1 \to \tau_2$, then $elim_{||}\ e_1\ e_2 : \tau_2$ under the condition that τ_2 is a mere proposition. Intuitively, propositional truncation hides the information in a kind of black box and lets you observe it only when computing a term which is itself "empty of information" (because it is a mere proposition).

3 A first attempt

In this section we will show a first attempt at defining a calculus with a kind of resizing axiom together with an encoding to and from a calculus with an impredicative bottom

$$
\begin{array}{ll}
||\cdot|| : \mathcal{U}_\ell \to \mathcal{U}_0 & \text{for all } \ell \in \mathbb{N} \\
|\cdot| \ : (t{:}\mathcal{U}_\ell) \to t \to ||t|| & \text{for all } \ell \in \mathbb{N} \\
\mathit{bind} : (t_1{:}\mathcal{U}_{\ell_1}) \to (t_2{:}\mathcal{U}_{\ell_2}) \to ||t_1|| \to (t_1 \to ||t_2||) \to ||t_2|| & \text{for all } \ell_1, \ell_2 \in \mathbb{N}
\end{array}
$$

$$
\frac{\Gamma \vdash \mathit{bind}\ \tau_1\ \tau_2\ |e_1|_{\tau_1}\ e_2\ :\ ||\tau_2||}{\Gamma \vdash \mathit{bind}\ \tau_1\ \tau_2\ |e_1|_{\tau_1}\ e_2\ \simeq\ e_2\ e_1\ :\ ||\tau_2||}\ (\beta_{||})
$$

Figure 4. Axioms of $\mathsf{r}_0\mathrm{CC}\omega$

universe. This is meant to show the general strategy we will use later on, but in a simpler setting, as well as illustrate some of the problems we encountered along the way and the way in which our resizing axioms have been refined, bringing them each time a bit closer to those used in HoTT.

3.1 The uCCω and r_0CCω calculi

Figure 3 shows our basic impredicative calculus we call $\mathsf{u}\mathrm{CC}\omega$, which consists in $\mathsf{p}\mathrm{CC}\omega$ modified to make its bottom universe impredicative. The result is a calculus comparable to the original Calculus of Constructions extended with a tower of universes, or seen another way, this is like Coq's core calculus stripped of all forms of (co)inductive types, cumulativity, and Set. Note that while this bottom universe is traditionally called Prop, we still call it $\mathcal{U}_0$.

Figure 4 shows the definitions we add to $\mathsf{p}\mathrm{CC}\omega$ in order to form $\mathsf{r}_0\mathrm{CC}\omega$, our first attempt at a calculus with a kind of resizing axiom. We can see that it introduces a new type constructor $||\cdot||$ (pronounced "erased"), along with an introduction form $|\cdot|_\tau$ (pronounced "erase" and where we will omit the type τ of its argument when it's clear from context), and an elimination form we called *bind* because this form of erasure forms a monad. The erasure $||\cdot||$ can be seen as a conflation of HoTT's propositional truncation with the propositional resizing, so rather than returning an erased version of the type in the same universe it immediately resizes it into the bottom universe $\mathcal{U}_0$. To bind the introduction and the elimination forms together we also included a conversion rule which is a form of β-reduction.

Note that this is *inspired* from HoTT's resizing axiom and propositional truncation but is much more limited, because we are trying to axiomatize the Prop universe, so we focus on the specific combination of those axioms that we need to use.

The use of a monad was partly inspired by a similar use of a monad to encode impredicativity by Spivack in its formalization of Hurkens's paradox in Coq [14]. It was also motivated by earlier failed efforts to solve this problem using the form of erasure found in ICC and EPTS [9, 2, 10], which does not form a monad, where it seemed that an operation like *bind* or *join* was an indispensable ingredient.

3.2 Encoding r_0CCω into uCCω

As a kind of warm up, we first try to encode terms of $\mathsf{r}_0\mathrm{CC}\omega$ into terms of $\mathsf{u}\mathrm{CC}\omega$. This turns out to be very easy because in $\mathsf{u}\mathrm{CC}\omega$ we can simply provide definitions for the axioms

of $\mathsf{r}_0\mathrm{CC}\omega$:

$$\begin{array}{l} ||\cdot|| \ : \ \mathcal{U}_\ell \to \mathcal{U}_0 \\ ||\tau|| \ = (t{:}\mathcal{U}_0) \to (\tau \to t) \to t \\ \\ |\cdot| \ : \ (t{:}\mathcal{U}_\ell) \to t \to ||t|| \\ |e|_\tau \ = \lambda t{:}\mathcal{U}_0.\lambda x{:}(\tau \to t).x\ e \\ \\ \mathit{bind} \ : \ (t_1{:}\mathcal{U}_{\ell_1}) \to (t_2{:}\mathcal{U}_{\ell_2}) \to ||t_1|| \to (t_1 \to ||t_2||) \to ||t_2|| \\ \mathit{bind} = \lambda t_1{:}\mathcal{U}_{\ell_1}.\lambda t_2{:}\mathcal{U}_{\ell_2}.\lambda x_1{:}||t_1||.\lambda x_2{:}(t_1 \to ||t_2||).x_1\ ||t_2||\ x_2 \end{array}$$

And we can easily verify that these definitions satisfy the convertibility rule (here and later as well, we will often omit the first two (type) arguments to *bind* to keep the code more concise) :

$$\begin{array}{l} \mathit{bind}\ |e_1|\ e_2 \\ \simeq |e_1|\ ||\tau_2||\ e_2 \\ \simeq (\lambda t{:}\mathcal{U}_0.\lambda x{:}(\tau_1 \to t).x\ e_1)\ ||\tau_2||\ e_2 \\ \simeq (\lambda x{:}(\tau_1 \to ||\tau_2||).x\ e_1)\ e_2 \\ \simeq e_2\ e_1 \end{array}$$

With these definitions in place, many properly typed term of $\mathsf{r}_0\mathrm{CC}\omega$ are also properly typed term (of the same type) of $\mathsf{u}\mathrm{CC}\omega$. Sadly, not all of them, because $\mathsf{u}\mathrm{CC}\omega$ is not a proper superset of $\mathsf{p}\mathrm{CC}\omega$: in $\mathsf{p}\mathrm{CC}\omega$ (and hence $\mathsf{r}_0\mathrm{CC}\omega$) a type like $(t{:}\mathcal{U}_0) \to t$ has type $\mathcal{U}_1$ whereas in $\mathsf{u}\mathrm{CC}\omega$ it can only have type $\mathcal{U}_0$.

3.3 Encoding $\mathsf{u}\mathrm{CC}\omega$ into $\mathsf{r}_0\mathrm{CC}\omega$

The other direction of the encoding cannot use the same trick of simply providing definitions. Instead we will translate terms with an encoding function $[\cdot]$. The core of the problem that we need to solve is that in $\mathsf{u}\mathrm{CC}\omega$, functions from $\mathcal{U}_\ell$ to $\mathcal{U}_0$ belong to universe $\mathcal{U}_0$ whereas in $\mathsf{r}_0\mathrm{CC}\omega$ they necessarily belong to universe $\mathcal{U}_\ell$, so the encoding will need to erase them with $||\cdot||$ in order to bring them down to $\mathcal{U}_0$.

Following the principle of Coq's Prop universe, which is proof-irrelevant, our encoding actually erases any and all elements of $\mathcal{U}_0$. The encoding function is basically syntax-driven, but it requires type information which is not directly available in the syntax of the terms, so technically the encoding takes as argument a typing *derivation*, but to make it more concise and readable, we write it as if its argument were just a term. Note that it does return just a term rather than a typing derivation. Here is our first attempt at encoding Prop into a resizing axiom :

$$\begin{array}{lll} [x] & = x & \\ [\mathcal{U}_\ell] & = \mathcal{U}_\ell & \\ [(x{:}\tau_1) \to \tau_2] & = \begin{cases} ||(x{:}[\tau_1]) \to [\tau_2]|| & \text{if in } \mathcal{U}_0 \\ (x{:}[\tau_1]) \to [\tau_2] & \text{otherwise} \end{cases} & \\ [\lambda x{:}\tau.e] & = \begin{cases} |\lambda x{:}[\tau].[e]| & \text{if in } \mathcal{U}_0 \\ \lambda x{:}[\tau].[e] & \text{otherwise} \end{cases} & \\ [e_1\ e_2] & = \begin{cases} \mathit{bind}\ [e_1]\ \lambda f{:}((x{:}[\tau_1]) \to [\tau_2]).f\ [e_2] & \text{if } e_1 \text{ in } \mathcal{U}_0 \\ [e_1]\ [e_2] & \text{otherwise} \end{cases} & \end{array}$$

A crucial property of such an encoding is type preservation : for any typing derivation $\Gamma \vdash e \ : \ \tau$ in $\mathsf{u}\mathrm{CC}\omega$, we need to show that there is a typing derivation $[\Gamma] \vdash [e] \ : \ [\tau]$ in $\mathsf{r}_0\mathrm{CC}\omega$. And the above encoding fails this basic test : the problem is that *bind* requires a return type of the form $||\tau_2||$ whereas in $\mathit{bind}\ [e_1]\ \lambda f{:}((x{:}[\tau_1]) \to [\tau_2]).f\ [e_2]$ the return type

$$
\begin{array}{rll}
\times & : \mathcal{U}_0 \to \mathcal{U}_0 \to \mathcal{U}_0 & \\
(\cdot,\cdot) & : (t_1{:}\mathcal{U}_0) \to (t_2{:}\mathcal{U}_0) \to t_1 \to t_2 \to t_1 \times t_2 & \\
\cdot.0 & : (t_1{:}\mathcal{U}_0) \to (t_2{:}\mathcal{U}_0) \to t_1 \times t_2 \to t_1 & \\
||\cdot|| & : \mathcal{U}_\ell \to \mathcal{U}_0 & \text{for all } \ell \in \mathbb{N} \\
|\cdot| & : (t{:}\mathcal{U}_\ell) \to t \to ||t|| & \text{for all } \ell \in \mathbb{N} \\
\mathsf{IsProp} & : \mathcal{U}_0 \to \mathcal{U}_0 & \\
\mathsf{isprop} & : (t{:}\mathcal{U}_\ell) \to \mathsf{IsProp}\ ||t|| & \text{for all } \ell \in \mathbb{N} \\
\mathit{elim}_{||} & : \begin{array}{l}(t_1{:}\mathcal{U}_\ell) \to (t_2{:}\mathcal{U}_0) \to \\ ||t_1|| \to (t_1 \to (t_2 \times \mathsf{IsProp}\ t_2)) \to (t_2 \times \mathsf{IsProp}\ t_2)\end{array} & \text{for all } \ell \in \mathbb{N}
\end{array}
$$

$$
\frac{\Gamma \vdash \mathit{elim}_{||}\ \tau_1\ \tau_2\ |e_1|_{\tau_1}\ e_2\ :\ \tau_2 \times \mathsf{IsProp}\ \tau_2}{\Gamma \vdash \mathit{elim}_{||}\ \tau_1\ \tau_2\ |e_1|_{\tau_1}\ e_2\ \simeq\ e_2\ e_1\ :\ \tau_2 \times \mathsf{IsProp}\ \tau_2}\ (\beta_{||}) \qquad \frac{\Gamma \vdash (e_1, e_2).0\ :\ \tau}{\Gamma \vdash (e_1, e_2).0\ \simeq\ e_1\ :\ \tau}\ (\beta.0)
$$

Figure 5. Axioms of rCCω

is $[\tau_2]$. This type is in the universe $\mathcal{U}_0$, so we know we will erase it, but as written, the types don't guarantee it. For example if τ_2 is a type variable t its encoding will just be t.

There is a very simple solution to this problem : change *bind* so it accepts any return type t_2. This would be compatible with our encoding, since our definition of *bind* in uCCω does not actually take advantage of the fact that the return type is erased. The problem is that it strengthens *bind* to the point of being too different from the $\mathit{elim}_{||}$ of HoTT : it would let us have a simple proof of equivalence between uCCω and r$_0$CCω but at the cost of making r$_0$CCω unrelated to the axiom of propositional resizing.

4 Encoding Prop as an axiom

In this section we analyze and fix the above problems, terminating with a proof of quasi-equivalence between uCCω and a new calculus rCCω.

Let us consider the following typing derivation in uCCω :

$$f_1{:}(\mathcal{U}_0 \to \mathcal{U}_0), t{:}\mathcal{U}_0, f_2{:}(t \to f_1\ t), x{:}t \vdash f_2\ x\ :\ f_1\ t$$

In order to be able to use *bind* in the encoding of $f_2\ x$, we need a proof that $[f_1\ t]$ will be an erased type. We can get this proof in one of two ways :

— We can obtain it from the encoding of $f_1\ t$ by making it so the encoding of a type that belongs to $\mathcal{U}_0$ is a pair of a type and a proof that it's erased.

— We can obtain it from the encoding of $f_2\ x$ by making it so the encoding of values in the bottom universe are pairs of a value and proof that this value has an erased type.

In either case we will want to adjust our axioms so *bind* does not require a return type of the form $||\tau_2||$ but is content with getting a proof that the return type is erased. To some extent, both can be made to work, but pairing the proof with the type requires a dependent pair, which we would not be able to encode into uCCω without extensions. So we will instead let f_2 return a value together with a proof that it has an erased type, since that only requires plain tuples which we can easily encode in uCCω.

Figure 5 shows the axioms of our new calculus rCCω. Compared to r$_0$CCω, we have added pairs (e_1, e_2) of type $\tau_1 \times \tau_2$, as well as a new predicate $\mathsf{IsProp}\ \tau$ with a single introduction form $\mathsf{isprop}\ \tau$ stating that $||\tau||$ satisfies this predicate. Furthermore *bind* is now renamed to $\mathit{elim}_{||}$ (since its type is not that of a monad's *bind* anymore) and it now requires the

$$
\begin{array}{lll}
[\![\tau]\!] & = \begin{cases} [\tau] \times \mathsf{IsProp}\ [\tau] & \text{if } \tau : \mathcal{U}_0 \\ [\tau] & \text{otherwise} \end{cases} & \\
[x] & = x & \\
[\mathcal{U}_\ell] & = \mathcal{U}_\ell & \\
[(x\!:\!\tau_1) \to \tau_2] & = \begin{cases} ||(x\!:\![\![\tau_1]\!]) \to [\![\tau_2]\!]|| & \text{if in } \mathcal{U}_0 \\ (x\!:\![\![\tau_1]\!]) \to [\![\tau_2]\!] & \text{otherwise} \end{cases} & \\
[\lambda x\!:\!\tau_1.e] & = \begin{cases} (|\lambda x\!:\![\![\tau_1]\!].[e]|, \mathsf{isprop}\ ((x\!:\![\![\tau_1]\!]) \to [\![\tau_2]\!])) & \text{if in } \mathcal{U}_0 \\ \lambda x\!:\![\![\tau]\!].[e] & \text{otherwise} \end{cases} & \\
[e_1\ e_2] & = \begin{cases} \mathit{elim}_{||}\ ([e_1].0)\ \lambda f\!:\!((x\!:\![\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2] & \text{if } e_1 : (x\!:\!\tau_1) \to \tau_2 : \mathcal{U}_0 \\ [e_1]\ [e_2] & \text{otherwise} \end{cases} &
\end{array}
$$

Figure 6. Encoding uCCω into rCCω

elimination to return a proof that the result is erased in the sense that it satisfies IsProp. Notice that we only included an elimination form to extract the first element of a pair but not the second and that there is no elimination form for $\mathsf{IsProp}\ \tau$. This is not an oversight but simply reflects the fact that our encoding does not directly make use of these eliminations, although they are presumably needed inside $\mathit{elim}_{||}$.

4.1 Encoding uCCω into rCCω

Figure 6 shows the new encoding function from uCCω into rCCω. The function is now split into two : the encoding of terms $[\cdot]$ and the encoding of types $[\![\cdot]\!]$. As before we abuse the notation in the sense that the functions as written seem to take only a syntactic term as argument, yet they really need more type information, such as the information that would come with a typing derivation as input. In a sense, instead of writing $[e]$ we should really write $[\Gamma \vdash e\ :\ \tau]$ and when we write $[\![\tau]\!]$ it similarly really means $[\![\Gamma \vdash \tau\ :\ \mathcal{U}_\ell]\!]$. An alternative would be to change the syntax of our terms so they come fully annotated everywhere with their types, or to make them use an intrinsically typed representation. But we opted for this abuse of notation because we feel that it lets the reader see the essence more clearly.

Note that both of those functions only return syntactic terms and not typing derivations. A mechanization of these functions might prefer to return typing derivations, so as to make it intrinsically type preserving, but for a paper proof like the one we present here, we found it preferable to return syntactic terms and then separately show the translation to be type preserving.

Lemma 4.1 (Substitution commutes with encoding).
If $\Gamma, x : \tau_2, \Gamma' \vdash e_1\ :\ \tau_1$ and $\Gamma \vdash e_2\ :\ \tau_2$ hold in uCCω, then in rCCω we have that $[e_1\{e_2/x\}] = [e_1]\{[e_2]/x\}$.

Proof sketch. By structural induction on the typing derivation of e_1. This is the direct consequence of the fact that $[x] = x$, which is an indispensable ingredient in all such syntactic models [3]. □

Lemma 4.2 (Computational soundness).
If $\Gamma \vdash e_1\ \simeq\ e_2\ :\ \tau$ holds in uCCω then $[\![\Gamma]\!] \vdash [e_1]\ \simeq\ [e_1]\ :\ [\![\tau]\!]$ holds in rCCω.

Proof sketch. This lemma needs to be proved by mutual induction with the lemma of type preservation since we need the types to be preserved in order to be able to instantiate the conversion rules in rCCω. An alternative would be to define our calculi with untyped conversion rules [13]. The proof also relies on the fact that $\Gamma \vdash e_1 \simeq e_2 : \tau$ implies both $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ in order to be able to use the $[\cdot]$ functions, although we omit the proof of this metatheoretical property which can be shown easily.

As for the proof itself : The cases for congruence rules are handled straightforwardly by the use of the induction hypothesis; For the β rule, we need to show that $[(\lambda x : \tau_1.e_1)\ e_2] \simeq [e_1\{e_2/x\}]$. The interesting case is when the function is in the universe $\mathcal{U}_0$, and hence erased :

$[(\lambda x{:}\tau_1.e_1)\ e_2]$
= [by definition of $[\cdot]$]
$\mathit{elim}_{||}\ ([(\lambda x{:}\tau_1.e_1)].0)\ \lambda f{:}((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2]$
= [by definition of $[\cdot]$]
$\mathit{elim}_{||}\ ((|\lambda x{:}[\![\tau_1]\!].[e_1]|, \mathsf{isprop}\ ((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!])).0)\ \lambda f{:}((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2]$
$\simeq$ [via the $\beta.0$ rule]
$\mathit{elim}_{||}\ (|\lambda x{:}[\![\tau_1]\!].[e_1]|)\ \lambda f{:}((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2]$
$\simeq$ [via the $\beta_{||}$ rule]
$(\lambda f{:}((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2])\ \lambda x{:}[\![\tau_1]\!].[e_1]$
$\simeq$ [via the β rule]
$(\lambda x{:}[\![\tau_1]\!].[e_1])\ [e_2]$
$\simeq$ [via the β rule]
$[e_1]\{[e_2]/x\}$
= [by the substitution lemma]
$[e_1\{e_2/x\}]$

□

Theorem 4.3 (Type Preserving encoding of uCCω into rCCω).
If we have $\Gamma \vdash e : \tau$ in uCCω, then $[\![\Gamma]\!] \vdash [e] : [\![\tau]\!]$ holds in rCCω.

Proof sketch. By induction on the typing derivation $\Gamma \vdash e : \tau$.

For the conversion rule, the proof defers all the work to the computational soundness lemma.

For the other rules, the more interesting case is the function application rule when the function is in $\mathcal{U}_0$ (i.e. the case that failed in our earlier naive attempt). In that case we have $\Gamma \vdash e_1\ e_2 : \tau_2\{e_2/x\}$ and we need to show

$$[\![\Gamma]\!] \vdash \mathit{elim}_{||}\ ([e_1].0)\ \lambda f{:}((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2] : [\![\tau_2\{e_2/x\}]\!]$$

By inversion we know that $\Gamma \vdash e_1 : (x{:}\tau_1) \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$. Hence by the induction hypothesis we have $[\![\Gamma]\!] \vdash [e_1] : [\![(x{:}\tau_1) \to \tau_2]\!]$ and $[\![\Gamma]\!] \vdash [e_2] : [\![\tau_1]\!]$. By definition of $[\![\cdot]\!]$ these rewrite to $[\![\Gamma]\!] \vdash [e_1] : ||(x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]|| \times \mathsf{IsProp}\ ||(x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]||$ and $[\![\Gamma]\!] \vdash [e_2] : [\tau_1] \times \mathsf{IsProp}\ [\tau_1]$.

Using the following shorthands :

$$\begin{aligned} P\ \tau &= \tau \times \mathsf{IsProp}\ \tau \\ T_1 &= (x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!] \end{aligned}$$

we can rewrite them as $[\![\Gamma]\!] \vdash [e_1] : P\ ||(x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]||$ or even $[\![\Gamma]\!] \vdash [e_1] : P\ ||T_1||$ and $[\![\Gamma]\!] \vdash [e_2] : P\ [\tau_1]$. Furthermore, since e_1 is in $\mathcal{U}_0$ we know that its return value is as well, so we know that $[\![\tau_2]\!] = P\ [\tau_2]$.

From that we get the desired conclusion using a mix of construction, weakening, and substitution :

$$\begin{array}{rcll} \mathcal{S} & = & \{\ \mathcal{U}_\ell & \mid\ \ell \in \mathbb{N}\ \} \\ \mathcal{A} & = & \{\ (\mathcal{U}_\ell : \mathcal{U}_{\ell+1}) & \mid\ \ell \in \mathbb{N}\ \} \\ \mathcal{R} & = & \{\ (\mathcal{U}_{\ell_1}, \mathcal{U}_{\ell_2}, \mathcal{U}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ \ell_1, \ell_2 \in \mathbb{N}\ \} \\ & & \cup\ \{\ (\mathcal{U}_\ell, \mathcal{U}_0, \mathcal{U}_0) & \mid\ \ell \in \mathbb{N}\ \} \end{array}$$

Figure 7. Definition of u⨣CCω as a PTS.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma]\!] \vdash [e_1]\ :\ P\ ||T_1||}}{[\![\Gamma]\!] \vdash [e_1]\ :\ ||T_1|| \times \ldots}}{[\![\Gamma]\!] \vdash [e_1].0\ :\ ||T_1||} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma]\!], f{:}T_1 \vdash f\ :\ T_1} \qquad \dfrac{}{[\![\Gamma]\!] \vdash [e_2]\ :\ [\![\tau_1]\!]}}{[\![\Gamma]\!], f{:}T_1 \vdash f\ [e_2]\ :\ (P\ [\tau_2])\{[e_2]/x\}}}{[\![\Gamma]\!], f{:}T_1 \vdash f\ [e_2]\ :\ P\ ([\tau_2]\{[e_2]/x\})}}{[\![\Gamma]\!], f{:}T_1 \vdash f\ [e_2]\ :\ P\ [\tau_2\{e_2/x\}]}}{[\![\Gamma]\!] \vdash \lambda f.f\ [e_2]\ :\ T_1 \to P\ [\tau_2\{e_2/x\}]}}{[\![\Gamma]\!] \vdash \mathit{elim}_{||}\ ([e_1].0)\ \lambda f.f\ [e_2]\ :\ P\ [\tau_2\{e_2/x\}]}}{[\![\Gamma]\!] \vdash \mathit{elim}_{||}\ ([e_1].0)\ \lambda f{:}((x{:}[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2]\ :\ [\![\tau_2\{e_2/x\}]\!]}$$

□

Theorem 4.4 (Consistency preservation of the encoding of uCCω into rCCω).
The type $[\![\bot]\!]$ is not inhabited in rCCω.

Proof sketch. This proof presumes without proof that rCCω is consistent. The traditional choice for $\bot$ would be $(x : \mathcal{U}_0) \to x$, but it is even simpler to use $(x : \mathcal{U}_1) \to x$, since $[\![(x{:}\mathcal{U}_1) \to x]\!]$ is just $(x{:}\mathcal{U}_1) \to x$ which should indeed not be inhabited in rCCω. □

Together those two theorems show the relative consistency of uCCω : if rCCω is consistent, then so is uCCω.

4.2 Encoding rCCω into u⨣CCω

Of course, now we still need to convert terms of our new calculus rCCω into uCCω. As we saw in Section 3, this tends to fail because of missing rules in the PTS of uCCω, so we will use u⨣CCω instead, which comes with the missing extra rules. Figure 7 shows the definition of u⨣CCω : the difference with uCCω is hidden in the first $\mathcal{R}$ rule where ℓ_2 is now allowed to be 0, which means that $(t{:}\mathcal{U}_0) \to t$ can now have type $\mathcal{U}_1$ as well as type $\mathcal{U}_0$, depending on whether we choose to use the first or the second rule. Contrary to uCCω and pCCω, u⨣CCω is not a *functional* PTS, which means that the universe inhabited by $(x : \tau_1) \to \tau_2$ is not uniquely determined by the universes inhabited by τ_1 and τ_2.

Now that we have a strong enough target language, Figure 8 shows how we convert rCCω into it using the same approach as for r_0CCω, i.e. by providing definitions for the various axioms.

The definitions are mostly unchanged, just supplemented with new ones. This is fundamentally the consequence of the fact that our definition of $\mathit{elim}_{||}$ does not actually need to look at the IsProp proof because our encoding of $||\cdot||$ lets us observe the "erased" term even if the result is not itself erased, as long as it is in $\mathcal{U}_0$. For this reason we can use degenerate definitions for our pairs and for IsProp.

As before, we have to make sure that those definitions satisfy the convertibility rules of rCCω. For $\beta_{||}$, the definition of $\mathit{elim}_{||}$ is basically the same as the earlier *bind*, so the conversion

$$
\begin{array}{lcl}
\cdot \times \cdot & : & \mathcal{U}_0 \to \mathcal{U}_0 \to \mathcal{U}_0 \\
t_1 \times t_2 & = & t_1 \\
\\
(\cdot,\cdot) & : & (t_1{:}\mathcal{U}_0) \to (t_2{:}\mathcal{U}_0) \to t_1 \to t_2 \to t_1 \times t_2 \\
(x_1, x_2) & = & x_1 \\
\\
\cdot.0 & : & (t_1{:}\mathcal{U}_0) \to (t_2{:}\mathcal{U}_0) \to t_1 \times t_2 \to t_1 \\
x.0 & = & x \\
\\
\mathsf{IsProp} & : & \mathcal{U}_0 \to \mathcal{U}_0 \\
\mathsf{IsProp}\ \tau & = & (t{:}\mathcal{U}_0) \to t \to t \\
\\
\mathsf{isprop} & : & (t{:}\mathcal{U}_\ell) \to \mathsf{IsProp}\ ||t|| \\
\mathsf{isprop}\ \tau & = & \lambda t{:}\mathcal{U}_0.\lambda x{:}t.x \\
\\
||\cdot|| & : & \mathcal{U}_\ell \to \mathcal{U}_0 \\
||\tau|| & = & (t{:}\mathcal{U}_0) \to (\tau \to t) \to t \\
\\
|\cdot| & : & (t{:}\mathcal{U}_\ell) \to t \to ||t|| \\
|e|_\tau & = & \lambda t{:}\mathcal{U}_0.\lambda x{:}(\tau \to t).x\ e \\
\\
elim_{||} & : & (t_1{:}\mathcal{U}_\ell) \to (t_2{:}\mathcal{U}_0) \to \\
 & & ||t_1|| \to (t_1 \to (t_2 \times \mathsf{IsProp}\ t_2)) \to (t_2 \times \mathsf{IsProp}\ t_2) \\
elim_{||} & = & \lambda t_1{:}\mathcal{U}_\ell.\lambda t_2{:}\mathcal{U}_0. \\
 & & \lambda x_1{:}||t_1||.\lambda x_2{:}(t_1 \to (t_2 \times \mathsf{IsProp}\ t_2)). \\
 & & x_1\ (t_2 \times \mathsf{IsProp}\ t_2)\ x_2
\end{array}
$$

Figure 8. Definitions for rCCω's axioms in uCCω

works just as before :

$$
\begin{array}{l}
elim_{||}\ |e_1|\ e_2 \\
\simeq |e_1|\ (\tau_2 \times \mathsf{IsProp}\ \tau_2)\ e_2 \\
\simeq (\lambda t{:}\mathcal{U}_0.\lambda x{:}(\tau_1 \to t).x\ e_1)\ (\tau_2 \times \mathsf{IsProp}\ \tau_2)\ e_2 \\
\simeq (\lambda x{:}(\tau_1 \to (\tau_2 \times \mathsf{IsProp}\ \tau_2)).x\ e_1)\ e_2 \\
\simeq e_2\ e_1
\end{array}
$$

And for $\beta.0$ it is even simpler, thanks to our degenerate encoding of pairs :

$$(e_1, e_2).0 \simeq e_1.0 \simeq e_1$$

Of course, a more traditional definition of pairs using Church's impredicative encoding would have worked as well.

We can put these definitions together in a substitution we will call σ_r. With these definitions in place, we can define our encoding as applying the substitution σ_r :

Theorem 4.5 (Type Preserving encoding of rCCω into u$^+$CCω).
If we have $\Gamma \vdash e\ :\ \tau$ in rCCω, then $\Gamma[\sigma_r] \vdash e[\sigma_r]\ :\ \tau[\sigma_r]$ in u$^+$CCω.

Proof sketch. Beside the axioms (provided by σ_r) and the new convertibility rules which we have just shown to be validated by σ_r, rCCω is a strict subset of u$^+$CCω. □

Theorem 4.6 (Consistency preservation of the encoding of rCCω into u$^+$CCω).
The encoding $\bot[\sigma_r]$ of rCCω's $\bot$ is not inhabited in u$^+$CCω.

Proof sketch. We presume again without proof that u$^+$CCω is consistent. Using $(x{:}\mathcal{U}_1) \to x$ as our $\bot$ again, we can see that $\bot$ does not refer to any of rCCω's axioms, so $((x{:}\mathcal{U}_1) \to x)[\sigma_r]$ is just $(x{:}\mathcal{U}_1) \to x$ which is indeed not inhabited in uCCω. □

$$\frac{\tau = \overrightarrow{(y:\tau_y)} \to \mathcal{U}_{\ell+1} \qquad \forall i.\ \ \Gamma, x:\tau \vdash \tau_i\ :\ \mathcal{U}_{\ell+1} \qquad \vdash \mathsf{isCon}(x, \tau_i)}{\Gamma \vdash \mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle\ :\ \tau}$$

$$\frac{\Gamma \vdash \tau_I\ :\ \tau \qquad \tau_I = \mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle}{\Gamma \vdash \mathsf{Con}(\tau_I, n)\ :\ \tau_n\{\tau_I/x\}}$$

$$\frac{\Gamma \vdash e\ :\ \tau_I\ \vec{\tau_u} \qquad \tau_I = \mathsf{Ind}(x:\_)\langle\vec{\tau}\rangle \qquad \forall i.\ \ \Gamma \vdash e_i\ :\ \Delta\{x, \tau_i, e_r, \mathsf{Con}(\tau_I, i)\}}{\Gamma \vdash \mathsf{Elim}(e, e_r)\langle\vec{e}\rangle\ :\ e_r\ \vec{\tau_u}\ e}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Elim}(\mathsf{Con}(\tau_I, i)\ \vec{e_s}, e_r)\langle\vec{e}\rangle\ :\ \tau \qquad \tau_I = \mathsf{Ind}(x:\overrightarrow{(x_x:\tau_x)} \to s)\langle\vec{\tau}\rangle \\ e_F = \lambda\overrightarrow{x_x:\tau_x}.\lambda x_c:\tau_I\ \vec{x_x}.\mathsf{Elim}(x_c, e_r)\langle\vec{e}\rangle\end{array}}{\Gamma \vdash \mathsf{Elim}(\mathsf{Con}(\tau_I, i)\ \vec{e_s}, e_r)\langle\vec{e}\rangle\ \simeq\ \Delta[x, \tau_i, e_i, e_F]\ \vec{e_s}\ :\ \tau}\ (\beta\text{-IND})$$

Figure 9. Main new rules for inductive types

Together those two theorems show the relative consistency of rCCω : if u$^+$CCω is consistent, then so is rCCω.

5 Inductive types

In this section we will show how this result generalizes to systems with inductive types in the higher, predicative universes, in the tradition of UTT [8]. We will call respectively uCCωI and rCCωI the previous calculi extended with inductive types. The extension uses the same syntax and typing rules in all calculi and is, in this sense, orthogonal to the differences in our underlying calculi.

5.1 Basic predicative inductive types

There are many different ways to define inductive types. We use here a presentation inspired from [17]. Nothing in this subsection is new material. Here is the extended syntax of the language :

$$\begin{array}{llll} \textit{(var)} & x, y, f, t & \in & \mathcal{V} \\ \textit{(sort)} & s & \in & \mathcal{S} \\ \textit{(term)} & e, \tau & ::= & s \mid x \mid (x:\tau_1) \to \tau_2 \mid \lambda x:\tau.e \mid e_1\ e_2 \\ & & & \quad \mid \mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle \mid \mathsf{Con}(\tau, n) \mid \mathsf{Elim}(e, e_r)\langle\vec{e}\rangle \end{array}$$

$\mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle$ is a new inductive type of kind τ where $\vec{\tau}$ are the types of its constructors, where x is bound (and refers to the inductive type itself) ; $\mathsf{Con}(\tau, n)$ is the n^{th} constructor of the inductive type τ ; and $\mathsf{Elim}(e, e_r)\langle\vec{e}\rangle$ is the corresponding eliminator, where e is a value of an inductive type, $\vec{e}$ are the branches corresponding to each one of the constructors of that type, and e_r is a function describing the return type of each branch and of the overall result. We use the notation $\vec{\tau}$ to mean 0 or more elements $\tau_0 ... \tau_n$; we use that same vector notation elsewhere to denote a (possibly empty) list of arguments.

Figure 9 shows the added rules of our language. These rules rely on auxiliary judgments shown in Figure 10. At the top are the three typing rules for the three new syntactic forms. The rule for Ind uses an auxiliary judgment $\vdash \mathsf{isCon}(x, \tau)$ which says that τ is a valid type for a constructor of an inductive type where x is a variable that stands for that inductive

$$\frac{x \notin \mathsf{fv}(\vec{e})}{\vdash \mathsf{isCon}(x, x\ \vec{e})} \qquad \frac{\vdash \mathsf{isCon}(x, \tau_2) \qquad x \notin \mathsf{fv}(\tau_y)}{\vdash \mathsf{isCon}(x, (y{:}\tau_y) \to \tau_2)}$$

$$\frac{\vdash \mathsf{isCon}(x, \tau_2) \qquad x \notin \mathsf{fv}(\overrightarrow{\tau_y}) \qquad x \notin \mathsf{fv}(\vec{e})}{\vdash \mathsf{isCon}(x, (\overrightarrow{(y{:}\tau_y)} \to x\ \vec{e}) \to \tau_2)}$$

$$\begin{array}{ll}
\Delta\{x, x\ \vec{e}, e_r, e_c\} & = e_r\ \vec{e}\ e_c \\
\Delta\{x, (y{:}\tau_y) \to \tau_2, e_r, e_c\} & = (y{:}\tau_y) \to \Delta\{x, \tau_2, e_r, e_c\ y\} \\
\Delta\{x, (\overrightarrow{(y{:}\tau_y)} \to x\ \vec{e}) \to \tau_2, e_r, e_c\} & = (x_p{:}(\overrightarrow{(y{:}\tau_y)} \to x\ \vec{e})) \to \\
& \quad (\overrightarrow{(y{:}\tau_y)} \to e_r\ \vec{e}\ (x_p\ \vec{y})) \to \\
& \quad \Delta\{x, \tau_2, e_r, e_c\ x_p\}
\end{array}$$

$$\begin{array}{ll}
\Delta[x, x\ \vec{e}, e_f, e_F] & = e_f \\
\Delta[x, (y{:}\tau_y) \to \tau_2, e_f, e_F] & = \lambda y{:}\tau_y.\Delta[x, \tau_2, e_f\ y, e_F] \\
\Delta[x, (\overrightarrow{(y{:}\tau_y)} \to x\ \vec{e}) \to \tau_2, e_f, e_F] & = (x_p{:}(\overrightarrow{(y{:}\tau_y)} \to x\ \vec{e})) \to \\
& \quad \Delta[x, \tau_2, e_f\ x_p\ (\lambda\overrightarrow{y{:}\tau_y}.e_F\ \vec{e}\ (e_p\ \vec{y})), e_F]
\end{array}$$

Figure 10. Auxiliary new rules for inductive types

type. This judgment thus verifies that τ indeed returns something of type x and that the only other occurrences of x in τ are in strictly positive positions. The rule for Con just extracts the type of the constructor from the inductive type itself. The rule for Elim enforces that it is applied to a value of an inductive type and checks that the type of each branch is consistent with the inductive type. To do that it relies on an auxiliary meta-level function $\Delta\{x, \tau, e_r, e_c\}$ which computes the type of a branch from the type τ of the corresponding constructor where e_r describe the return type of the elimination, and e_c is a reconstruction of the value being matched by the branch. This function is basically defined by induction on the $\vdash \mathsf{isCon}(x, \tau)$ proof that the constructor's type is indeed valid. You can see in the second line of that definition that for every field of the inductive type, the branch gets a corresponding argument (the field's value) and in the third line you can see that in addition to that, for those fields which hold a recursive value the branch receives the result of performing the induction on that field.

The final rule of Figure 9 shows the new reduction rule for inductive types. The term e_F defined there represents a recursive call to the eliminator, which is applied to every recursive field of the constructor. Like the typing rule of Elim, this rule uses an auxiliary meta-function $\Delta[x, \tau, e_f, e_F]$ which computes the appropriate call to the branch e_f from the type τ of the constructor, and where e_F is the function to use to recurse. Just like $\Delta\{x, \tau, e_r, e_c\}$, this function is basically defined by induction on the $\vdash \mathsf{isCon}(x, \tau)$ proof that the constructor's type is valid.

5.2 From uCCωI to rCCωI

When encoding terms from uCCωI into rCCωI, we can actually keep the same rules as those we used when encoding uCCω into rCCω except of course that we need to add new rules for the new constructs. The result is shown in Figure 11.

Since our inductive types all live in universes above $\mathcal{U}_0$, their encoding is straightforward. Yet, as we can see in the case of Elim, it requires some care : the last two arguments e_r and $\vec{e}$ to Elim are functions that need to be treated specially :

$$
\begin{array}{lll}
[\![\tau]\!] & = \begin{cases} [\tau] \times \mathsf{IsProp}\ [\tau] & \text{if } \tau : \mathcal{U}_0 \\ [\tau] & \text{otherwise} \end{cases} & \\
{[x]} & = x & \\
[\mathcal{U}_\ell] & = \mathcal{U}_\ell & \\
[(x:\tau_1) \to \tau_2] & = \begin{cases} ||(x:[\![\tau_1]\!]) \to [\![\tau_2]\!]|| & \text{if in } \mathcal{U}_0 \\ (x:[\![\tau_1]\!]) \to [\![\tau_2]\!] & \text{otherwise} \end{cases} & \\
[\lambda x:\tau_1.e] & = \begin{cases} (|\lambda x:[\![\tau_1]\!].[e]|, \mathsf{isprop}\ ((x:[\![\tau_1]\!]) \to [\![\tau_2]\!])) & \text{if in } \mathcal{U}_0 \\ \lambda x:[\![\tau]\!].[e] & \text{otherwise} \end{cases} & \\
[e_1\ e_2] & = \begin{cases} \mathit{elim}_{||}\ ([e_1].0)\ \lambda f:((x:[\![\tau_1]\!]) \to [\![\tau_2]\!]).f\ [e_2] & \text{if } e_1 : (x:\tau_1) \to \tau_2 : \mathcal{U}_0 \\ [e_1]\ [e_2] & \text{otherwise} \end{cases} & \\
{[\mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle]} & = \mathsf{Ind}(x:[\![\tau]\!])\langle[\vec{\tau}]\rangle & \\
{[\mathsf{Con}(\tau, n)]} & = \mathsf{Con}([\tau], n) & \\
{[\mathsf{Elim}(e, e_r)\langle\vec{e}\rangle]} & = \mathsf{Elim}([e], e'_r)\langle\vec{e'}\rangle & \\
 & \quad \text{where } \Gamma \vdash e\ :\ \tau_I\ \vec{\tau_u} & \\
 & \qquad \tau_I = \mathsf{Ind}(x:\tau')\langle\vec{\tau}\rangle & \\
 & \qquad \tau' = \overrightarrow{(y:\tau_y)} \to \mathcal{U}_{\ell+1} & \\
 & \qquad e'_r = \lambda \overrightarrow{y:[\![\tau_y]\!]}.\lambda x_d:[\![\tau_I\ \vec{y}]\!].[\![e_r\ \vec{y}\ x_d]\!] & \\
 & \qquad e'_i = \mathsf{EtaBranch}(x, \tau', e_i) & \\
 & \qquad \mathsf{EtaBranch}(x, x\ \vec{e}, e_b) = [e_b] & \\
 & \qquad \mathsf{EtaBranch}(x, (y:\tau_y) \to \tau_2, e_b) = \lambda y:[\![\tau_y]\!].\mathsf{EtaBranch}(x, \tau_2, e_b\ y) & \\
 & \qquad \mathsf{EtaBranch}(x, (\overrightarrow{(y:\tau_y)} \to x\ \vec{e_x}) \to \tau_2, e_b) & \\
 & \qquad\quad = \lambda x_p:\left[\!\!\left[\overrightarrow{(y:\tau_y)} \to x\ \vec{e_X}\right]\!\!\right].\lambda z:\left[\!\!\left[\overrightarrow{(y:\tau_y)} \to e_r\ \vec{e_x}\ (x_p\ \vec{y})\right]\!\!\right]. & \\
 & \qquad\qquad \mathsf{EtaBranch}(x, \tau_2, e_b\ x_p\ z) &
\end{array}
$$

Figure 11. Encoding uCCωI into rCCωI

— Since e_r represents the return *type* of the elimination (and of its branches), it crucially needs to be encoded with $[\![\cdot]\!]$ rather than with $[\cdot]$, yet it is not a type but a function which returns a type, so we need to η-expand it in order to be able to apply $[\![\cdot]\!]$ to the type it returns rather than to the function itself.
— $\vec{e}$ holds the code of each of the branches in the form of functions, but this use of functions is mostly incidental, basically a kind of HOAS (Higher-Order Abstract Syntax [12]) encoding of the syntax. Depending on the return type of the elimination those functions may belong to the $\mathcal{U}_0$ universe yet we should not erase them since Elim would then not know what to do with them. So, similarly to what we did for e_r, we need to η-expand those functions in order to encode the body rather than the function itself.

Lemma 5.1 (Substitution commutes with encoding).
If $\Gamma, x : \tau_2, \Gamma' \vdash e_1 \ : \ \tau_1$ and $\Gamma \vdash e_2 \ : \ \tau_2$ hold in uCCωI, then in rCCωI we have that $[e_1\{e_2/x\}] = [e_1]\{[e_2]/x\}$.

Proof sketch. This still holds, as before, by structural induction on the typing derivation of e_1. □

Lemma 5.2 (Computational soundness).
If $\Gamma \vdash e_1 \ \simeq \ e_2 \ : \ \tau$ holds in uCCωI then $[\![\Gamma]\!] \vdash [e_1] \ \simeq \ [e_1] \ : \ [\![\tau]\!]$ holds in rCCωI.

Proof sketch. The proof from the previous section mostly carries over unchanged. In the new reduction rule for Elim, the η-expansions we applied to the branches of Elim make life more difficult, but they get β-reduced as they should. □

Lemma 5.3 (Positivity preservation).
If we have $\tau = \overrightarrow{(y : \tau_y)} \to \mathcal{U}_{\ell+1}$ and $\Gamma, x : \tau \vdash \tau_i \ : \ \mathcal{U}_{\ell+1}$ and $\vdash \mathsf{isCon}(x, \tau)$ in uCCωI, then we also have $\vdash \mathsf{isCon}(x, [\tau])$ in rCCωI.

Proof sketch. This holds by structural induction because τ lives in a universe above $\mathcal{U}_0$ so the encoding does not touch the overall structure of τ, only the type of individual arguments and only the non-recursive arguments, so the positivity is not affected. □

Theorem 5.4 (Type Preserving encoding of uCCωI into rCCωI).
If we have $\Gamma \vdash e \ : \ \tau$ in uCCωI, then $[\![\Gamma]\!] \vdash [e] \ : \ [\![\tau]\!]$ holds in rCCωI.

Proof sketch. As was the case for computational soundness, the proof steps from the previous section carry over unchanged. For Ind, we need to use the positivity preservation lemma. The Con case is trivial. The Elim case is tedious but does not present any particular difficulty. □

Theorem 5.5 (Consistency preservation of the encoding of uCCωI into rCCωI).
The type $[\![\bot]\!]$ is not inhabited in rCCωI.

Proof sketch. Unchanged. □

Theorem 5.6 (Consistency preservation of the encoding of rCCωI into u$\dagger$CCωI).
The encoding $\bot[\sigma_r]$ of rCCωI's $\bot$ is not inhabited in u$\dagger$CCωI.

Proof sketch. Indeed, rCCωI uses exactly the same extra axioms as rCCω did, so we use the same σ_r as before and the proof holds unchanged. □

6 Applicability

The quasi-equivalence we have shown between $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ and $\mathsf{r}\mathrm{CC}\omega\mathrm{I}$ seems to confirm our initial intuition that an impredicative Prop universe is closely related to the propositional resizing axiom. But there are still important differences to reconcile, starting with the difference between $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ and $\mathsf{u}^{+}\mathrm{CC}\omega\mathrm{I}$, as well as between our axioms and the propositional resizing axiom, and important differences between $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ and a system like Coq :

— The difference between $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ and $\mathsf{u}^{+}\mathrm{CC}\omega\mathrm{I}$ may seem minor since it is usually perfectly safe to *lift* a type from one universe to a higher one, as can happen for example in systems that use universe subsumption. This said, encoding $\mathsf{u}^{+}\mathrm{CC}\omega$ into $\mathsf{u}\mathrm{CC}\omega$ seems difficult since there is no way in $\mathsf{u}\mathrm{CC}\omega$ to lift types from $\mathcal{U}_0$ to a higher universe. We would need to add axioms for it. The situation is more promising in $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ where we can wrap elements from $\mathcal{U}_0$ into an inductive type to lift them to a higher universe. Still, figuring out where to insert those lifts is left for future work.
In a sense this is related to the question in Coq of distinguishing those definitions which *rely* on impredicativity from those that just happen to be defined in Prop but could work just as well with a predicative universe. Except here, it's the reverse : the encoding from $\mathsf{r}\mathrm{CC}\omega\mathrm{I}$ falls within the $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ subset of $\mathsf{u}^{+}\mathrm{CC}\omega\mathrm{I}$ for those $\mathcal{U}_0$ elements that rely on impredicativity (and hence used the resizing axiom) but not for the other ones, so we need to find those other ones and make them use Prop even though they do not need its impredicativity.
Another avenue would be to try and encode $\mathsf{u}^{+}\mathrm{CC}\omega$ into $\mathsf{r}\mathrm{CC}\omega$, but the syntactic approach we use is difficult to use with non-functional pure type systems because a type like $(x : \tau_1) \to \tau_2$ can be at the same time in $\mathcal{U}_0$ and in some other universe, making it difficult to decide how to encode it.
— Our $||\cdot||$ erasure axiom combines propositional truncation and propositional resizing in a single indivisible step. It is not clear what is the impact of this conflation but it is an important difference : we cannot resize without truncating at the same time, contrary to HoTT.
— Our inductive types live in universes above $\mathcal{U}_0$. This is probably the main limitation of our current work. Handling inductive types in the impredicative universe poses several important challenges : the main one is that to support strong elimination of small inductive types in $\mathcal{U}_0$ we cannot apply $||\cdot||$ to those inductive types, which means we cannot apply $||\cdot||$ to all terms in $\mathcal{U}_0$ any more. Maybe a solution might be to stay closer to HoTT's propositional truncation which allows elimination into other universes than $\mathcal{U}_0$.
— Our calculi enjoy only the usual β laws but not the η laws. It seems quite challenging to extend this work to allow $e_1 \simeq \lambda x.e_1\ x$. Maybe quotient types can provide a solution, but it appears impossible to give a definitional equality to the quotient types we would need, so we would have to manipulate equality proofs, making the overall structure of the encoding significantly more complex, probably degenerating to something similar to the encoding of ETT in ITT [18].
— The difference between $\mathsf{u}\mathrm{CC}\omega\mathrm{I}$ and $\mathsf{u}^{+}\mathrm{CC}\omega\mathrm{I}$ is not the only reason why what we show is not a strict equivalence : the two transformations are not inverses of each other. Since we are mostly concerned about consistency, expressiveness, and interactions with additional axioms, we do not consider it as a significant weakness, but it is a weakness nevertheless.

7 Conclusion

In the previous installment of our investigation into impredicativity [11], we had a look at how it relates to the kind of erasure found in systems like EPTS and ICC [9, 2, 10], which

is a different kind of erasure from that associated with proof irrelevance.

This time, we have shown a close relationship between the two main ways to provide impredicativity in current type theories, namely via a Prop impredicative universe, or via a propositional resizing axiom, both of which are associated with proof irrelevance.

Amusingly, in our previous paper we were able to extend its base result from CCω to a calculus extended with inductive types in the bottom (impredicative) universe, as found in the original CIC papers [17], but not to a calculus with inductive types in the higher universes, like UTT. This time, in contrast, we were able to extend our result from CCω to a calculus like UTT but not to a calculus with inductive types in the bottom universe.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N^o 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

Références

[1] Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2) :121–154, April 1991. doi:10.1017/S0956796800020025.

[2] Bruno Barras and Bruno Bernardo. Implicit calculus of constructions as a programming language with dependent types. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, Budapest, Hungary, April 2008. doi:10.1007/978-3-540-78499-9_26.

[3] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs*, page 182–194, 2017. doi:10.1145/3018610.3018620.

[4] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78, August 2009. doi:10.1007/978-3-642-03359-9_6.

[5] Thierry Coquand and Gérard P. Huet. The calculus of constructions. Technical Report RR-0530, INRIA, 1986.

[6] Gérard P. Huet, Christine Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

[7] Antonius Hurkens. A simplification of Girard's paradox. In *International conference on Typed Lambda Calculi and Applications*, pages 266–278, 1995. doi:10.1007/BFb0014058.

[8] Zhaohui Luo. A unifying theory of dependent types : the schematic approach. In *Logical Foundations of Computer Science*, 1992. doi:10.1007/BFb0023883.

[9] Alexandre Miquel. The implicit calculus of constructions : extending pure type systems with an intersection type binder and subtyping. In *International conference on Typed Lambda Calculi and Applications*, pages 344–359, 2001. URL : https://www.fing.edu.uy/~amiquel/publis/tlca01.pdf, doi:10.1007/3-540-45413-6_27.

[10] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364, Budapest, Hungary, April 2008. URL : https://web.cecs.pdx.edu/~sheard/papers/FossacsErasure08.pdf, doi:10.1007/978-3-540-78499-9_25.

[11] Stefan Monnier and Nathaniel Bos. Is impredicativity implicitly implicit ? In *Types for Proofs and Programs*, Leibniz International Proceedings in Informatics (LIPIcs), pages 9 :1–9 :19, 2019. doi:10.4230/LIPIcs.TYPES.2019.9.

[12] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Programming Languages Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[13] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(2) :153–180, March 2012. doi:10.1017/S0956796812000044.

[14] Arnaud Spiwack. Notes on axiomatising hurkens's paradox, 2015. URL : https://arxiv.org/abs/1507.04577.

[15] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL : https://arxiv.org/abs/1308.0729.

[16] Vladimir Voevodsky. Resizing rules - their use and semantic justification. Slides from a talk in Bergen., sep 2011. URL : https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf.

[17] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994. URL : https://hal.inria.fr/tel-00196524/.

[18] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating reflection from type theory. In *CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. URL : https://hal.science/hal-01849166.