Resizing Prop down to an axiom

Stefan Monnier
Université de Montréal - DIRO, Montréal, Canada

Abstract

Impredicativity and type theory have 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 by making the bottom universe impredicative, typically called Prop; the other way, proposed by Voevodsky [15] uses axioms that allow moving some types from one universe to a lower one, the main example of those being the propositional resizing axiom as found in HoTT [14].

While Coq's Prop and HoTT's propositional resizing axiom seem intuitively closely related, since they both restrict the use of impredicative quantification to the definition of proof irrelevant propositions, 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 proving equivalence between specific calculi with either an impredicative bottom universe or a set of axioms closely related to the propositional resizing axiom. We first show it for a pure type system with the usual infinite tower of universes, and then we extend this result with the addition of inductive types.

This final result shows as a first approximation that the kind of impredicativity provided by Coq's Prop universe is virtually identical to that offered by HoTT's propositional resizing. In practical terms, the syntactic nature of the proof of equivalence means that it can be used also to translate code between such systems.

2012 ACM Subject Classification Theory of computation → Type theory; Software and its engineering → Functional languages; Theory of computation → Higher order logic; Theory of computation → Constructive mathematics

Keywords and phrases Impredicativity, Pure type systems, Inductive types , Proof irrelevance, Resizing axiom , Proof system interoperability

Digital Object Identifier 10.4230/LIPIcs...

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N^o 298311/2012 and RGPIN-2018-06225.

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, 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.

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 [15] 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 axioms is the *propositional resizing axiom* used in Homotopy type theory [14].

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 are closely 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 precisely how they compare by proving equivalence between a calculus using a Prop universe and one using a resizing axiom.

Our contributions are:

- A proof of equivalence between iCCω, an impredicative pure type system with a tower of universes, and rCCω, its sibling based on the predicative subset pCCω extended with a variant of the propositional resizing axiom.
- An extension of that proof to calculi with inductive types iCICω and rCICω. The complexity of this extension depends on the details of how inductive types are introduced. To cover the kind of definitions allowed in Coq, the extension requires a slightly refined resizing axiom.
- The proofs of equivalence 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 show it to be equivalent to iCCω; 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 with related works.

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

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

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

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

Figure 1 Main typing rules of our PTS

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 $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ 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:

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

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 $(\Box, *, *)$

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

Figure 2 Definition of pCCω as a PTS.

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 Type_ℓ where ℓ is called the universe level and Type_0 is the bottom universe. To keep things simple, our universes are not cumulative, although our development would work just as well in the presence of cumulative universes.

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 $id = \Lambda t.\lambda x\!:\!t.x$ is quantified over any type t, including the type $\forall t.t \to t$ of the polymorphic identity. 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 $(\mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\mathsf{max}(\ell_1,\ell_2)})$, 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 $(\mathsf{Type}_\ell, \mathsf{Type}_0, \mathsf{Type}_0)$ which allow impredicative quantifications in the *bottom* universe Type_0. Such an impredicative bottom universe is traditionally called Prop.

2.2 Propositional resizing

In Homotopy type theory [14], 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:

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

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

Figure 3 Definition of iCCω as a PTS.

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

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

Figure 4 Axioms of r_0CCω

The resizing axiom says that any type which is a mere proposition in a universe $\mathsf{Type}_{\ell+1}$ can be "resized" to an equivalent one in the smaller universe Type_ℓ. By repeated application, it follows that any mere proposition can be resized to belong to the bottom universe Type_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 $\mathit{elim}_{||}$) which says that if $|e_1| : ||\tau_1||$ and $e_2 : \tau_1 \to \tau_2$, then $\mathit{elim}_{||}\ e_1\ e_2 : \tau_2$ 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 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 iCCω and r_0CCω calculi

Figure 3 shows our basic impredicative calculus we call iCCω, which consists in pCCω extended with the traditional rules making 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 inductive types. Note that while this bottom universe is traditionally called Prop, we still call it Type_0.

Figure 4 shows the definitions we add to pCCω in order to form r_0CCω, 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 often omit the τ), 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 return an erased version of the type in the same universe it immediately resizes it into the bottom universe Type_0. To bind the introduction and the elimination forms together we also included a conversion rule which is a form of β reduction.

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 [13]. It was also motivated by earlier failures to solve this problem we encountered when 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 $\mathsf{r}_0\mathsf{CC}\omega$ into $\mathsf{iCC}\omega$

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

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

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):

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

With these definitions in place, any properly typed term of $\mathsf{r}_0\mathrm{CC}\omega$ is also a properly typed term (of the same type) of $\mathsf{iCC}\omega$.

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

The other direction of the encoding cannot use the same trick. Instead we will translate terms with an encoding function $[\cdot]$. The core of the problem that we need to solve is that in $\mathsf{iCC}\omega$, functions from Type_ℓ to Type_0 can belong to universe Type_0 whereas in $\mathsf{r}_0\mathrm{CC}\omega$ they necessarily belong to universe Type_ℓ, so the encoding will need to erase them with $||\cdot||$ in order to bring them down to Type_0.

Following the principle of Coq's Prop universe, which is proof-irrelevant, our encoding actually erases any and all elements of Type_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:

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

A crucial property of such an encoding is type preservation: for any typing derivation $\Gamma \vdash e\ :\ \tau$ in iCCω, we need to show that there is a typing derivation $[\Gamma] \vdash [e]\ :\ [\tau]$ in r$_0$CCω. 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 is $[\tau_2]$. This type is in the universe Type_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 iCCω 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 iCCω 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 problem, terminating with a proof of equivalence between iCCω and rCCω.

Let us consider the following typing derivation in iCCω:

$$f_1:(\mathsf{Type}_0 \to \mathsf{Type}_0), t:\mathsf{Type}_0, f_2:(t \to f_1\ t), x:t \vdash f_2\ x\ :\ f_1\ t$$

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 Type_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 iCCω without extensions. So we will instead let

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

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

Figure 5 Axioms of rCCω

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

Figure 6 Encoding iCCω into rCCω

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 iCCω.

Figure 5 shows the axioms of our new calculus rCCω. Compared to r_0CCω, 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 stating that $||\tau||$ satisfies this predicate. Furthermore *bind* is now renamed to $\mathit{elim}_{||}$ (since it does not quite fit the monad shape any more) and it now requires the 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 iCCω into rCCω

Figure 6 shows the new encoding function from iCCω 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 only take 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 $[\![\Gamma \vdash \tau \ : \ \mathsf{Type}_\ell]\!]$. 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 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 iCCω, then in rCCω we have that $[e_1\{e_2/x\}] = [e_1]\{[e_2]/x\}$.

Proof. 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 2** (Computational soundness).
If $\Gamma \vdash e_1 \ \simeq \ e_2 \ : \ \tau$ holds in iCCω then $[\![\Gamma]\!] \vdash [e_1] \ \simeq \ [e_1] \ : \ [\![\tau]\!]$ holds in rCCω.

Proof. 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 [12]. 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 congruence rules are straightforward. 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

Type_0:

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

◀

▶ **Theorem 3** (Type Preserving encoding of iCCω into rCCω)**.** If we have $\Gamma \vdash e\ :\ \tau$ in iCCω, then $[\![\Gamma]\!] \vdash [e]\ :\ [\![\tau]\!]$ holds in rCCω.

Proof. 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 Type_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:

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

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 Type_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:

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

Figure 7 Definitions for rCCω's axioms in iCCω

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

◀

▶ **Theorem 4** (Consistency preservation of the encoding of iCCω into rCCω)**.** The type $[\![\bot]\!]$ is not inhabited in rCCω.

Proof. The traditional choice for $\bot$ would be $(x : \mathsf{Type}_0) \to x$, but we will use $(x : \mathsf{Type}_1) \to x$, since $[\![(x : \mathsf{Type}_1) \to x]\!]$ is just $(x : \mathsf{Type}_1) \to x$ which should indeed not be inhabited in rCCω. ◀

4.2 Encoding rCCω into iCCω

Of course, now we still need to make sure that we can convert terms of our new calculus rCCω into iCCω. Figure 7 shows how we do this using the same approach as for r_0CCω, i.e. by providing definitions for the various axioms.

We note that our definition of $elim_{||}$ 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 Type_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 $elim_{||}$ is basically the same as the earlier *bind*, so the conversion works just as before:

$$
\begin{aligned}
& elim_{||}\ |e_1|\ e_2 \\
& \simeq |e_1|\ (\tau_2 \times \mathsf{IsProp}\ \tau_2)\ e_2 \\
& \simeq (\lambda t{:}\mathsf{Type}_0.\lambda x{:}(\tau_1 \to t).x\ e_1)\ (\tau_2 \times \mathsf{IsProp}\ \tau_2)\ e_2 \\
& \simeq (\lambda x{:}(\tau_1 \to (\tau_2 \times \mathsf{IsProp}\ \tau_2)).x\ e_1)\ e_2 \\
& \simeq e_2\ e_1
\end{aligned}
$$

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 5** (Type Preserving encoding of rCCω into iCCω).
If we have $\Gamma \vdash e\ :\ \tau$ in rCCω, then $\Gamma[\sigma_r] \vdash e[\sigma_r]\ :\ \tau[\sigma_r]$ in iCCω.

Proof. 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 iCCω. ◀

▶ **Theorem 6** (Consistency preservation of the encoding of rCCω into iCCω).
The encoding $\bot[\sigma_r]$ of rCCω's $\bot$ is not inhabited in iCCω.

Proof. Using $(x : \mathsf{Type}_1) \to x$ as our $\bot$ again, we can see that $\bot$ does not refer to any of rCCω's axioms, so $((x : \mathsf{Type}_1) \to x)[\sigma_r]$ is just $(x : \mathsf{Type}_1) \to x$ which is indeed not inhabited in iCCω. ◀

5 Inductive types

As the degenerate definitions in the previous section suggest, limiting ourselves to pure type systems like iCCω does not exercise the full complexity of modern impredicative systems. In this section we will show how to extend the previous result to systems with inductive types, which we will call respectively iCICω and rCICω.

The first thing to note is that we can take the systems from the previous section and add inductive types in higher universes (i.e. Type_ℓ for $\ell > 0$), as was done in UTT [8], and the

$$\frac{\tau = \overrightarrow{(y:\tau_y)} \to s \qquad \forall i.\ \ \Gamma, x:\tau \vdash \tau_i\ :\ s \qquad \vdash \mathsf{isCon}(x, \tau_i)}{\Gamma \vdash \mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle\ :\ \tau} \qquad \frac{\tau = \mathsf{Ind}(x:\tau')\langle\vec{\tau}\rangle}{\Gamma \vdash \mathsf{Con}(\tau, n)\ :\ \tau_n\{\tau/x\}}$$

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

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

Figure 8 Main new rules of pCICω

previous results will carry over trivially, since the encodings leave all the entities from higher universes basically untouched.

Things get interesting only once we try to add inductive types in Type_0. For example, inductive types in rCICω would normally have no restrictions when it comes to their elimination rules, including for strong elimination. Of course, having fully predicative universes, an inductive type in rCICω only lives in Type_0 if it's so-called "small", i.e. it only carries values which themselves live in Type_0. In the original CIC, such as presented in [17], such small types also supported arbitrary strong elimination, but this corresponds to Coq's impredicative Set universe, which does not enjoy proof-irrelevance and hence seems to be impossible to encode into a system with a propositional resizing axiom. The kind of impredicative universe we can hope to encode using a resizing axiom would be Coq's Prop universe, where strong elimination of small inductive types is restricted to those small types that only have a single constructor, so that they can be erased. This in turn means that encoding from rCICω to iCICω will not be as simple as before: iCICω is not just a strict superset of pCICω.

In the other direction we also encounter new difficulties: if our encoding erases all iCICω terms in Type_0 like we did in the previous section, then strong elimination of those erased inductive types will be problematic since those eliminations will not themselves return an erased value.

5.1 Basic predicative inductive types: pCICω

Before defining iCICω and rCICω we start by extending pCCω with inductive types, to have a shared starting point pCICω from which to define them. There are many different ways to define inductive types. We use here a presentation inspired from [17]. Nothing in this subsection is new. Here is the extended syntax of the language:

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

$\mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle$ 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

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

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

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

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

Figure 9 Auxiliary new rules of pCICω

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 8 shows the added rules of our language. These rules rely on auxiliary judgments shown in Figure 9. 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 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 we induce on 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 see in that definition that for every field of the inductive type, the branch gets a corresponding argument (the field's value) and 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 8 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 Impredicative universe and inductive types: iCICω

As mentioned, since we intend to encode the impredicativity of iCICω using a kind of propositional resizing axiom, we will not try to provide a proof-relevant impredicative universe like Coq's impredicative Set but we will instead make our bottom universe proof irrelevant like Coq's Prop.

The first step is as before: we add new impredicative quantification rules. This time, rather than make Type_0 impredicative, we add a new Prop universe underneath all others:

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

We also need to adjust the rules of inductive types to make sure this new Prop universe is proof-irrelevant and to avoid introducing inconsistencies. We do this by refining the typing rule of Elim as follows:

$$\frac{\Gamma \vdash e\ :\ \tau_I\ \vec{\tau_u} \qquad \tau_I = \mathsf{Ind}(x : \overrightarrow{(x_x : \tau_x)} \to s_I)\langle\vec{\tau}\rangle \qquad \forall i.\quad \Gamma \vdash e_i\ :\ \Delta\{x, \tau_i, e_r, \mathsf{Con}(\tau_I, i)\} \\ \Gamma \vdash e_r\ \vec{\tau_u}\ e\ :\ s_r \qquad s_r = \mathsf{Prop} \vee s_I = \mathsf{Type}_\ell \vee (|\vec{\tau}| \leq 1\ \wedge\ \Gamma \vdash \mathsf{isSmall}(\vec{\tau}))}{\Gamma \vdash \mathsf{Elim}(e, e_r)\langle\vec{e}\rangle\ :\ e_r\ \vec{\tau_u}\ e}$$

where $\Gamma \vdash \mathsf{isSmall}(\tau)$ makes sure that all the fields of this constructor belong to the Prop universe. The extra side conditions are meant to rule out strong eliminations of large inductive types, because they render the system inconsistent, and the additional $|\vec{\tau}| = 1$ is the check that makes sure that terms of the Prop universe are proof-irrelevant (i.e. can be erased).

5.3 rCICω

5.4 From iCICω to rCICω

$$
\begin{array}{lll}
\mathsf{Erase}\ \tau & = \langle ||\tau||, \mathsf{isprop}\ \tau \rangle & \\
\llbracket \tau \rrbracket & = \begin{cases} \Sigma t.\mathsf{IsProp}\ t & \text{if } \tau = \mathsf{Prop} \\ [\tau].0 & \text{if } \tau : \mathsf{Prop} \\ [\tau] & \text{otherwise} \end{cases} & \\
\\
[x] & = x & \\
[\mathsf{Prop}] & = \mathsf{Type}_0 & \\
[\mathsf{Type}_\ell] & = \mathsf{Type}_{\ell+1} & \\
[(x:\tau_1) \to \tau_2] & = \begin{cases} \mathsf{Erase}((x:\llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket) & \text{if in Prop} \\ (x:\llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket & \text{otherwise} \end{cases} & \\
[\lambda x:\tau.e] & = \begin{cases} |\lambda x:\llbracket\tau_1\rrbracket.[e]| & \text{if in Prop} \\ \lambda x:\llbracket\tau\rrbracket.[e] & \text{otherwise} \end{cases} & \\
[e_1\ e_2] & = \begin{cases} elim_{||}\ ((x:\llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket)\ \llbracket\tau_2\rrbracket\ ([\tau_2].1) & \text{if } e_1 \text{ in Prop} \\ \quad [e_1]\ \lambda f:((x:\llbracket\tau_1\rrbracket) \to \llbracket\tau_2\rrbracket).f\ [e_2] & \\ [e_1]\ [e_2] & \text{otherwise} \end{cases} & \\
\\
[\overrightarrow{(x:\tau)} \to s]_I & = \overrightarrow{(x:\llbracket\tau\rrbracket)} \to [s] & \\
[x\ \vec{e}]_c & = x\ \vec{[e]} & \\
[(y:\tau_y) \to \tau]_c & = (y:\llbracket\tau_y\rrbracket) \to [\tau]_c & \\
[(\overrightarrow{(y:\tau_y)} \to x\ \vec{e}) \to \tau]_c & = (\overrightarrow{(y:\llbracket\tau_y\rrbracket)} \to x\ \vec{[e]}) \to [\tau]_c & \\
[\mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle] & = \begin{cases} \overrightarrow{\lambda x_I:\tau_I}.\mathsf{Erase}((\mathsf{Ind}(x:[\tau]_I)\langle\vec{[\tau]_c}\rangle)\ \vec{x_I}) & \\ \qquad \text{if } \mathsf{needErase}(\mathsf{Ind}(x:\tau)\langle\vec{\tau}\rangle) & \\ \langle(\mathsf{Ind}(x:[\tau]_I)\langle\vec{[\tau]_c}\rangle), & \text{if in Prop} \\ \ \mathsf{isprop}\ \ (\mathsf{Ind}(x:[\tau]_I)\langle\vec{[\tau]_c}\rangle)\rangle & \\ \mathsf{Ind}(x:[\tau]_I)\langle\vec{[\tau]_c}\rangle & \text{otherwise} \end{cases} & \\
\\
[\mathsf{Con}(\tau, n)] & = \begin{cases} \overrightarrow{\lambda x:\tau}.|\mathsf{Con}([\tau], n)\ \vec{x}| & \text{if } \mathsf{needErase}(\tau) \\ \mathsf{Con}([\tau], n) & \text{otherwise} \end{cases} &
\end{array}
$$

6 Applicability

7 Related works and conclusion

[13] Uses a similar monad to represent impredicativity.

[3] [16] [5] [14] [11]

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.

References

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. 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. 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.

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

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

15. 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.

450 **16** Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A.
451 Eisenberg. A specification for dependent types in Haskell. In *International Conference on*
452 *Functional Programming*, page 1–29, 2017. doi:10.1145/3110275.

453 **17** Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université
454 Paris 7, Paris, France, 1994. URL: https://hal.inria.fr/tel-00196524/.