Resizing Prop down to an axiom

Stefan Monnier
Université de Montréal - DIRO, Montréal, Canada

Abstract
Impredicativity and type theory have a long history since Russell 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 No 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.
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:

\[
\begin{align*}
(x, f, t) & \in V \\
 s & \in S \\
e, \tau & ::= s \mid x \mid (x : \tau_1) \to \tau_2 \mid \lambda x : \tau. e \mid e_1 e_2
\end{align*}
\]

Terms can be either a sort \( s \); or a variable \( x \); or a function \( \lambda x : \tau. e \) where \( x \) is the formal argument, \( \tau \) 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 \( \tau_1 \) is the type of the argument and \( \tau_2 \) is the type of the result and where \( x \) is bound within \( \tau_2 \). We can write \( \tau_1 \to \tau_2 \) if \( x \) does not occur in \( \tau_2 \). A specific PTS is then defined by providing the tuple \( (S, A, R) \) which defines respectively the set \( S \) of sorts, the axioms \( A \) that relate the various sorts, and the rules \( R \) 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 \( \tau \) in context \( \Gamma \). We have two auxiliary judgments: \( \vdash \Gamma \) says that \( \Gamma \) 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 \( \tau \) in context \( \Gamma \). 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:

\[
S = \{ *, \Box \} \\
A = \{ (* : \Box) \} \\
R = \{ (*, *, *), (\Box, *, *) \}
\]

Where \(*\) is the universe of values and \(\Box\) is the universe of types and the axiom \((*, *)\) expresses the fact that types classify values. The rule \((*, *, *)\) corresponds to the traditional “small \(\lambda\)” 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 \((\Box, *, *)\)
Resizing Prop down to an axiom

\[ S = \{ \text{Type}_\ell \mid \ell \in \mathbb{N} \} \]
\[ A = \{ (\text{Type}_\ell : \text{Type}_{\ell + 1}) \mid \ell \in \mathbb{N} \} \]
\[ R = \{ (\text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\max(\ell_1, \ell_2)}) \mid \ell_1, \ell_2 \in \mathbb{N} \} \]

**Figure 2** Definition of pCCω as a PTS.

 corresponds to the traditional “big A” and says that functions can also quantify over “types” (i.e. elements of the universe □) 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_\ell where \ell 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 \((□, *, *)\) is the source of impredicativity because it allows the creation of a function in * which quantifies over elements that belong to the larger universe □ and which can hence include its own type. To make it predicative, we would need to use \((□, *, □)\) meaning that a function that quantifies over types and returns values would now belong to the universe □. 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 \( id[\mathbb{N} \to \mathbb{N}][id[\mathbb{N}]]. \)

In contrast to System-F, we can see that pCCω is predicative because its rules have the form \((\text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\max(\ell_1, \ell_2)})\), 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 \((\text{Type}_\ell, \text{Type}_0, \text{Type}_0)\) 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:

\[ \text{isProp } \tau : (x : \tau) \to (y : \tau) \to x = y \]
\[ S = \{ \text{Type}_\ell : \ell \in \mathbb{N} \} \]
\[ A = \{ \text{Type}_\ell : \text{Type}_{\ell+1} : \ell \in \mathbb{N} \} \]
\[ \mathcal{R} = \{ (\text{Type}_\ell_1, \text{Type}_\ell_2, \text{Type}_{\max(\ell_1, \ell_2)} ) : \ell_1, \ell_2 \in \mathbb{N} \} \]

**Figure 3** Definition of $\operatorname{iCC}_{\omega}$ as a PTS.

\[ || \cdot || : \text{Type}_\ell \rightarrow \text{Type}_0 \ \ \ \text{for all} \ \ \ell \in \mathbb{N} \]
\[ | \cdot | : (t : \text{Type}_\ell) \rightarrow t \rightarrow ||t|| \ \ \ \text{for all} \ \ \ell \in \mathbb{N} \]
\[ \text{bind} : (t_1 : \text{Type}_\ell) \rightarrow (t_2 : \text{Type}_\ell) \rightarrow ||t_1|| \rightarrow (t_1 \rightarrow ||t_2||) \rightarrow ||t_2|| \ \ \ \text{for all} \ \ \ell_1, \ell_2 \in \mathbb{N} \]

**Figure 4** Axioms of $\operatorname{r}_{0}\operatorname{CC}_{\omega}$

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

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 \(\text{elim}_{||}\)) which says that if $|e_1| : ||\tau_1||$ and $e_2 : \tau_1 \rightarrow \tau_2$, then $\text{elim}_{||} e_1 e_2 : \tau_2$ under the condition that $\tau_2$ 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 iCC$_{\omega}$ and r$_{0}\operatorname{CC}_{\omega}$ calculi

Figure 3 shows our basic impredicative calculus we call iCC$_{\omega}$, which consists in pCC$_{\omega}$ 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 pCC$_{\omega}$ in order to form r$_{0}\operatorname{CC}_{\omega}$, our first attempt at a
Resizing Prop down to an axiom

calculus with a kind of resizing axiom. We can see that it introduces a new type constructor
\(|·]|\) (pronounced “erased”), along with an introduction form \(|·|_{\tau}\) (pronounced “erase” and
where we will often omit the \(\tau\), and an elimination form we called \(\text{bind}\) because this form of
erasure forms a monad. The erasure \(|·]|\) 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 \(\text{Type}_{0}\). To bind
the introduction and the elimination forms together we also included a conversion rule which
is a form of \(\beta\) 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 \(\text{bind}\) or \(\text{join}\) was an indispensable ingredient.

3.2 Encoding \(r_{0}\text{CC}\omega\) into \(i\text{CC}\omega\)

As a kind of warm up, we first show how we can encode any term of \(r_{0}\text{CC}\omega\) into a term of
\(i\text{CC}\omega\). This turns out to be very easy because in \(i\text{CC}\omega\) we can simply provide definitions for
the axioms of \(r_{0}\text{CC}\omega\):
\[
|·| : \text{Type}_{\ell} \rightarrow \text{Type}_{0}
\]
\[
|\tau|| : (t : \text{Type}_{0}) \rightarrow (\tau \rightarrow t) \rightarrow t
\]
\[
|·|_{\tau} : (t : \text{Type}_{\ell}) \rightarrow t \rightarrow |t||
\]
\[
|e|_{\tau} = \lambda t_{1} : \text{Type}_{\ell}. \lambda x : (\tau \rightarrow t). x e
\]
\[
\text{bind} : (t_{1} : \text{Type}_{\ell_{1}}) \rightarrow (t_{2} : \text{Type}_{\ell_{2}}) \rightarrow |t_{1}|| \rightarrow (t_{1} \rightarrow |t_{2}||) \rightarrow |t_{2}||
\]
\[
\text{bind} = \lambda t_{1} : \text{Type}_{\ell_{1}}. \lambda t_{2} : \text{Type}_{\ell_{2}}. \lambda x_{1} : |t_{1}||. \lambda x_{2} : (t_{1} \rightarrow |t_{2}||). x_{1} |t_{2}|| x_{2}
\]

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 \(\text{bind}\) to keep the code more
concise):
\[
\text{bind} |e_{1}| e_{2} \approx |e_{1}| |t_{2}|| e_{2}
\]
\[
\approx (\lambda t_{1} : \text{Type}_{\ell_{1}}. \lambda x : (t_{1} \rightarrow t). x e_{1}) |t_{2}|| e_{2}
\]
\[
\approx (\lambda x : (t_{1} \rightarrow |t_{2}||). x e_{1}) e_{2}
\]
\[
\approx e_{2} e_{1}
\]

With these definitions in place, any properly typed term of \(r_{0}\text{CC}\omega\) is also a properly typed
term (of the same type) of \(i\text{CC}\omega\).

3.3 Encoding \(i\text{CC}\omega\) into \(r_{0}\text{CC}\omega\)

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
\(i\text{CC}\omega\), functions from \(\text{Type}_{\ell}\) to \(\text{Type}_{0}\) can belong to universe \(\text{Type}_{0}\), whereas in \(r_{0}\text{CC}\omega\) they
necessarily belong to universe \(\text{Type}_{\ell}\), so the encoding will need to erase them with \(|·|\) in
order to bring them down to \(\text{Type}_{0}\).

Following the principle of Coq’s \(\text{Prop}\) universe, which is proof-irrelevant, our encoding actually
erases any and all elements of \(\text{Type}_{0}\). 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:
\[
\begin{align*}
[x] & = x \\
[\text{Type}_i] & = \text{Type}_i \\
[(x: \tau_1) \rightarrow \tau_2] & = \begin{cases} 
\{((x: [\tau_1]) \rightarrow [\tau_2]) \} & \text{if in Type}_0 \\
\{ (x: [\tau_1]) \rightarrow [\tau_2] \} & \text{otherwise}
\end{cases}
\end{align*}
\]
\[
[\lambda x: \tau.e] = \begin{cases} 
[\lambda x: [\tau],[e]] & \text{if in Type}_0 \\
\lambda x: [\tau],[e] & \text{otherwise}
\end{cases}
\]
\[
[e_1 \ e_2] = \begin{cases} 
\text{bind } [e_1] \lambda f: ((x: [\tau_1]) \rightarrow [\tau_2]).f \ [e_2] & \text{if } e_1 \text{ in Type}_0 \\
[e_1] \ [e_2] & \text{otherwise}
\end{cases}
\]

A crucial property of such an encoding is type preservation: for any typing derivation \(\Gamma \vdash e : \tau\) in \(i\text{CC}\omega\), we need to show that there is a typing derivation \([\Gamma] \vdash [e] : [\tau]\) in \(r_0\text{CC}\omega\). And the above encoding fails this basic test: the problem is that bind requires a
return type of the form \([\tau_2]\) whereas in bind \([e_1]\) \(\lambda f: ((x: [\tau_1]) \rightarrow [\tau_2]).f \ [e_2]\) the return type
is \([\tau_2]\). This type is in the universe \(\text{Type}_0\), so we know we will erase it, but as written, the
types don’t guarantee it. For example if \(\tau_2\) 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 \(\tau_2\).
This would be compatible with our encoding, since our definition of bind in \(i\text{CC}\omega\) 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 elim| of HoTT: it would let
us have a simple proof of equivalence between \(i\text{CC}\omega\) and \(r_0\text{CC}\omega\) but at the cost of making
\(r_0\text{CC}\omega\) 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 \(i\text{CC}\omega\) and \(r\text{CC}\omega\).

Let us consider the following typing derivation in \(i\text{CC}\omega\):
\[f_1: (\text{Type}_0 \rightarrow \text{Type}_0), t: \text{Type}_0, f_2: (t \rightarrow f_1 \ t), x: t \vdash f_2 \ x : f_1 \ t\]
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 \(i\text{CC}\omega\) without extensions. So we will instead let
Resizing Prop down to an axiom

× : Type₀ → Type₀ → Type₀
(·,·) : (t₁ : Type₀) → (t₂ : Type₀) → t₁ → t₂ → t₁ × t₂
·.0 : (t₁ : Type₀) → (t₂ : Type₀) → t₁ × t₂ → t₁

|| · || : Typeₖ → Type₀
| · | : (t : Typeₖ) → t → ||t|| for all ℓ ∈ ℕ

IsProp : Type₀ → Type₀
isprop : (t : Typeₖ) → IsProp ||t|| for all ℓ ∈ ℕ
elim || : (t₁ : Typeₖ) → (t₂ : Type₀) → ||t₁|| → (t₁ → (t₂ × IsProp t₂)) → (t₂ × IsProp t₂) for all ℓ ∈ ℕ

Γ ⊢ elim || τ₁ τ₂ e₁ e₂ : τ₂ × IsProp τ₂
Γ ⊢ (e₁, e₂).0 : τ (β₁ ||)
Γ ⊢ (e₁, e₂).0 ≃ e₁ : τ (β₀)

Figure 5 Axioms of rCCω

[[τ]] = \begin{cases} [τ] \times IsProp [τ] & \text{if } τ : Type₀ \\ [τ] & \text{otherwise} \end{cases}

[x] Typeₖ = Typeₖ

[(x : τ₁) → τ₂] = \begin{cases} ||(x : [[τ₁]]) → [[τ₂]]|| & \text{if in Type₀} \\ (x : [[τ₁]]) → [[τ₂]] & \text{otherwise} \end{cases}

[λx : τ₁.c] = \begin{cases} (|λx : [[τ₁]].[c]|, isprop ((λx : [[τ₁]].[c])) & \text{if in Type₀} \\ λx : [[τ]], [c] & \text{otherwise} \end{cases}

[e₁ e₂] = \begin{cases} elim || ((e₁).0) λf : ((x : [[τ₁]]) → [[τ₂]]).f [e₂] & \text{if } e₁ : (x : τ₁) → τ₂ : Type₀ \\ e₁ [e₂] & \text{otherwise} \end{cases}

Figure 6 Encoding iCCω into rCCω

f₂ 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 τ₀CCω, we have added pairs (e₁, e₂) of type τ₁ × τ₂, as well as a new predicate IsProp τ with a single introduction form stating that ||τ|| satisfies this predicate. Furthermore bind is now renamed to 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 IsProp τ. 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 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 \([\Gamma \vdash \tau : \text{Type}_\ell]\). 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 \(\Gamma \vdash e_1 \equiv e_2 : \tau\) holds in iCCω then \([\Gamma] \vdash [e_1] \equiv [e_1] : [\tau]\) 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 \(\Gamma \vdash e_1 \equiv e_2 : \tau\) 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 \(\beta\) rule, we need to show that \([(\lambda x : \tau_1.e_1) \ e_2] \equiv [e_1 \{e_2/x\}]\). The interesting case is when the function is in
Resizing Prop down to an axiom

We can rewrite them as

\[ \text{Type}_0: \]

\[
\begin{align*}
[(\lambda x: \tau_1). e_2] \\
= [\text{by definition of } \lambda] \\
\text{elim}_1 \left( ((\lambda x: \tau_1). e_1), 0 \right) \lambda f: ((x: \tau_1) \rightarrow \tau_2). f \ [e_2] \\
= [\text{by definition of } \lambda] \\
\text{elim}_1 \left( ((\lambda x: \tau_1). e_1), 0 \right), \text{isprop} \left( ((x: \tau_1) \rightarrow \tau_2), 0 \right) \lambda f: ((x: \tau_1) \rightarrow \tau_2). f \ [e_2] \\
\simeq [\text{via the } \beta_0 \text{ rule}] \\
\text{elim}_1 \left( ((\lambda x: \tau_1). e_1), 0 \right) \lambda f: ((x: \tau_1) \rightarrow \tau_2). f \ [e_2] \\
\simeq [\text{via the } \lambda \text{ rule}] \\
\lambda x: \tau_1, e_1 \ [e_2] \\
\simeq [\text{via the } \beta \text{ rule}] \\
[e_1] \ [\{e_2/x\}]
\end{align*}
\]

\[ [\text{by the substitution lemma}] \]

\[ [e_1 \ [e_2/x]] \]

---

**Theorem 3** (Type Preserving encoding of \( \text{iCC} \omega \) into \( \text{rCC} \omega \)).

If we have \( \Gamma \vdash e : \tau \) in \( \text{iCC} \omega \), then \( [\Gamma] \vdash [e] : [\tau] \) holds in \( \text{rCC} \omega \).

**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 \( \text{Type}_0 \) (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

\[ [\Gamma] \vdash \text{elim}_1 \left( [e_1], 0 \right) \lambda f: ((x: \tau_1) \rightarrow \tau_2). f \ [e_2] : [\tau_2 \ [e_2/x]] \]

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

\[ [\Gamma] \vdash [e_2] : [\tau_1] \times \text{IsProp} \ [\tau_1] \]

Using the following shorthands:

\[ P \ \tau = \tau \times \text{IsProp} \ \tau \]

\[ \text{r_1} = (x: [\tau_1]) \rightarrow [\tau_2] \]

we can rewrite them as \( [\Gamma] \vdash [e_1] : P \ [(x: [\tau_1]) \rightarrow [\tau_2]] \) or even \( [\Gamma] \vdash [e_1] : P \ [r_1] \) and

\[ [\Gamma] \vdash [e_2] : P \ [\tau_1] \]

Furthermore, since \( e_1 \) is in \( \text{Type}_0 \) 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:
\[
\begin{align*}
\cdot \times \cdot & : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \\
t_1 \times t_2 & = t_1 \\
(\cdot, \cdot) & : ((t_1 : \text{Type}) \rightarrow (t_2 : \text{Type}) \rightarrow t_1 \rightarrow t_2 \rightarrow t_1 \times t_2 \\
(x_1, x_2) & = x_1 \\
\cdot.0 & : (t_1 : \text{Type}) \rightarrow (t_2 : \text{Type}) \rightarrow t_1 \times t_2 \rightarrow t_1 \\
x.0 & = x \\
\text{IsProp} & : \text{Type} \rightarrow \text{Type} \\
\text{IsProp} \ t & = (t : \text{Type}) \rightarrow t \rightarrow t \\
isprop & : (t : \text{Type}) \rightarrow \text{IsProp} \ ||t|| \\
isprop \ t & = \lambda t : \text{Type}, \lambda x : (t \rightarrow t).x \ e \\
|| \cdot || & : \text{Type} \rightarrow \text{Type} \\
||t|| & = (t : \text{Type}) \rightarrow (t \rightarrow t) \rightarrow t \\
| \cdot | & : (t : \text{Type}) \rightarrow t \rightarrow ||t|| \\
|e|_t & = \lambda t : \text{Type}, \lambda x : (t \rightarrow t).x \ e \\
\text{elim}_|| & : (t_1 : \text{Type}) \rightarrow (t_2 : \text{Type}) \rightarrow \\
& \quad \quad ||t_1|| \rightarrow (t_1 \rightarrow (t_2 \times \text{IsProp} \ t_2)) \rightarrow (t_2 \times \text{IsProp} \ t_2) \\
\text{elim}_| & = \lambda t_1 : \text{Type}, \lambda t_2 : \text{Type}.
\lambda x_1 : ||t_1||, \lambda x_2 : (t_1 \rightarrow (t_2 \times \text{IsProp} \ t_2)). \\
& \quad \quad x_1 (t_2 \times \text{IsProp} \ t_2) \ x_2
\end{align*}
\]

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{definitions.png}
\caption{Definitions for rCCω’s axioms in iCCω}
\end{figure}

\textbf{Theorem 4} (Consistency preservation of the encoding of iCCω into rCCω).

The type \([\bot]\) is not inhabited in rCCω.

\textbf{Proof.} The traditional choice for \(\bot\) would be \((x : \text{Type}) \rightarrow x\), but we will use \((x : \text{Type}) \rightarrow x\), since \([\bot](x : \text{Type}) \rightarrow x\) is just \((x : \text{Type}) \rightarrow x\) which should indeed not be inhabited in rCCω. 
\hspace{1cm} \blacksquare
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 r0CCω, i.e. by providing definitions for the various axioms.

We note that our definition of \(\text{elim}_||\) does not actually need to look at the IsProp proof because our encoding of \(||·||\) lets us observe the “erased” term even if the result is not itself erased, as long as it is in Type0. 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 \(\text{elim}_||\) is basically the same as the earlier bind, so the conversion works just as before:

\[
\text{elim}_|| (e_1,e_2) \simeq (e_1,\text{IsProp }\tau) e_2 \\
\simeq (\lambda t : \text{Type }0. \lambda x : (\tau_1 \rightarrow (\tau_2 \times \text{IsProp }\tau_2)). x e_1) e_2 \\
\simeq e_2 e_1
\]

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 \(\sigma_r\). With these definitions in place, we can define our encoding as applying the substitution \(\sigma_r\):

\[\text{Theorem 5} \ (\text{Type Preserving encoding of rCCω into iCCω}).\]

If we have \(\Gamma \vdash e : \tau\) in rCCω, then \(\Gamma[\sigma_r] \vdash e[\sigma_r] : \tau[\sigma_r]\) in iCCω.

\[\text{Proof.}\] Beside the axioms (provided by \(\sigma_r\)) and the new convertibility rules which we have just shown to be validated by \(\sigma_r\), rCCω is a strict subset of iCCω. ◀

\[\text{Theorem 6} \ (\text{Consistency preservation of the encoding of rCCω into iCCω}).\]

The encoding \(\bot[\sigma_r]\) of rCCω’s \(\bot\) is not inhabited in iCCω.

\[\text{Proof.}\] Using \((x : \text{Type}_1) \rightarrow x\) as our \(\bot\) again, we can see that \(\bot\) does not refer to any of rCCω’s axioms, so \(((x : \text{Type}_1) \rightarrow x)[\sigma_r]\) is just \((x : \text{Type}_1) \rightarrow \bot\) 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. \(\text{Type}_\ell\) for \(\ell > 0\)), as was done in UTT [8], and the
\[
\begin{align*}
\tau = (y: \tau_0) \rightarrow s & \quad \forall i. \quad \Gamma, x: \tau \vdash \tau_i : s \quad \vdash \text{isCon}(x, \tau_i) \\
& \quad \vdash \text{Ind}(x: \tau)(\vec{\tau}) \\
& \quad \vdash \text{Con}(\tau, n) : \tau_n(\tau/x) \\
\end{align*}
\]

\[\begin{align*}
\Gamma \vdash e : \tau_I \tau^n_u & \quad \tau_I = \text{Ind}(x: \_)(\vec{\tau}) \\
& \quad \forall i. \quad \Gamma \vdash e_i : \Delta\{x, \tau_i, e_r, \text{Con}(\tau_I, i)\} \\
& \quad \vdash \text{Elim}(e_i, e_r)(\vec{\tau}) : e_r \tau^n_u e \\
\end{align*}\]

\[\begin{align*}
\Gamma \vdash \text{Elim}(\text{Con}(\tau_I, i) \ e_s, e_r)(\vec{\tau}) : \tau & \quad \tau_I = \text{Ind}(x: \tau(x) \rightarrow s)(\vec{\tau}) \\
& \quad e_F = \lambda x_s: \tau^n_u. \lambda x_c: \tau_I. \ x_s.x_c.\text{Elim}(x_c, e_r)(\vec{\tau}) \\
& \quad \vdash \text{Elim}(\text{Con}(\tau_I, i) \ e_s, e_r)(\vec{\tau}) \simeq \Delta[x, \tau_i, e_i, e_F] \ e_s : \tau \\
& \quad (\beta - \text{Ind})
\end{align*}\]

**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 Type0. 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 Type0 if it’s so-called “small”, i.e. it only carries values which themselves live in Type0. 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 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 Type0 like we did in the previous section, then strong elimination of those erased inductive types 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 pCICω 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:

\[
\begin{align*}
\text{(var)} & \quad x, y, f, t \in \mathcal{V} \\
\text{(sort)} & \quad s \in \mathcal{S} \\
\text{(term)} & \quad e, \tau ::= s \mid x \mid (x: \tau_1) \rightarrow \tau_2 \mid \lambda x.e \mid e_1. e_2 \mid \text{Ind}(x: \tau)(\vec{\tau}) \mid \text{Con}(\tau, n) \mid \text{Elim}(e, e_r)(\vec{\tau})
\end{align*}
\]

\(\text{Ind}(x: \tau)(\vec{\tau})\) is a new inductive type of kind \(\tau\) where \(\vec{\tau}\) are the types of its constructors, where \(x\) is bound (and refers to the inductive type itself); \(\text{Con}(\tau, n)\) is the \(n^{th}\) constructor of the inductive type \(\tau\); and \(\text{Elim}(e, e_r)(\vec{\tau})\) is the corresponding eliminator, where \(e\) is a value of
Resizing Prop down to an axiom

\[
\begin{align*}
\Delta\{x, \vec{r}, e_r, e_c\} &= e_r \vec{r} e_c \\
\Delta\{x, ((y:\tau_y) \to x, \vec{r}) \to \tau_2, e_r, e_c\} &= (y:\tau_y) \to \Delta\{x, \tau_2, e_r, e_c\} y \\
\Delta\{x, ((y:\tau_y) \to x, \vec{r}) \to \tau_2, e_r, e_c\} &= (x_p:(y:\tau_y) \to x, \vec{r}) \to \\
&\quad \Delta\{x, \tau_2, e_r, e_c\} x_p (\lambda y:\tau_y, e_F \vec{r} (e_p y)) (\vec{r}(x_p y)) e_F \\
\end{align*}
\]

\textbf{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 \(e_c\) is a function describing the return type of each branch and of the overall result.

We use the notation \(\tau\) 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 \texttt{Ind} uses an auxiliary judgment \(\vdash \text{isCon}(x, \tau)\) which says that \(\tau\) 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 \(\tau\) indeed returns something of type \(x\) and that the only other occurrences of \(x\) in \(\tau\) are in strictly positive positions. The rule for \texttt{Con} just extracts the type of the constructor from the inductive type itself. The rule for \texttt{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 \(\tau\) 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 \(\vdash \text{isCon}(x, \tau)\) 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 \texttt{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 \(\tau\) 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 \(\vdash \text{isCon}(x, \tau)\) proof that the constructor’s
type is valid.

5.2 Impredicative universe and inductive types: iCIC\(\omega\)

As mentioned, since we intend to encode the impredicativity of iCIC\(\omega\) 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:

\[
\begin{align*}
S &= \{ \text{Prop}, \text{Type}_\ell \mid \ell \in \mathbb{N} \} \\
A &= \{ (\text{Prop}:\text{Type}_0), (\text{Type}_\ell : \text{Type}_{\ell+1}) \mid \ell \in \mathbb{N} \} \\
\mathcal{R} &= \{ (\text{Type}_{\ell_1}, \text{Type}_{\ell_2}, \text{Type}_{\max(\ell_1, \ell_2)}) \mid \ell_1, \ell_2 \in \mathbb{N} \} \\
&\quad \cup \{ (\text{Prop}, \text{Type}_\ell) \mid \ell \in \mathbb{N} \} \\
&\quad \cup \{ (s, \text{Prop}, \text{Prop}) \}
\end{align*}
\]

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 \(\text{Elim}\) as follows:

\[
\begin{align*}
\Gamma \vdash e : \tau_I \quad \tau_u \quad \tau_f &= \text{Ind}(x:(x:x:\tau_u) \to s_I)(\bar{\tau}) \\
&\quad \forall i. \quad \Gamma \vdash e_i : \Delta(x, \tau_1, \tau_r, \text{Con}(\tau_I, i)) \\
\Gamma \vdash e_r \quad \tau_u \quad e : \text{Prop} \lor s_I = \text{Type}_\ell \lor (|\bar{\tau}| \leq 1 \land \Gamma \vdash \text{isSmall}(\bar{\tau})) \\
\Gamma \vdash \text{Elim}(e, e_r)(\bar{e}) : e_r \quad \tau_u \quad e
\end{align*}
\]

where \(\Gamma \vdash \text{isSmall}(\tau)\) 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 \(|\bar{\tau}| = 1\) 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

Erase τ = ⟨]|| τ ||, isprop τ⟩

[τ] = \begin{cases} \Sigma t . \text{IsProp } t & \text{if } τ = \text{Prop} \\ [τ].0 & \text{if } τ : \text{Prop} \\ [τ] & \text{otherwise} \end{cases}

[x] = x

[Prop] = Type₀

[Type] = Type₁

[(x : τ₁) → τ₂] = \begin{cases} \text{Erase}((x : [τ₁] → [τ₂])) & \text{if in } \text{Prop} \\ (x : [τ₁]) → [τ₂] & \text{otherwise} \end{cases}

[λx : τ. e] = \begin{cases} \text{elim|} (\text{Erase}((x : [τ₁] → [τ₂])) [τ₂].1) & \text{if } e_1 \text{ in } \text{Prop} \\ \text{λ} x : [τ]. [e] & \text{otherwise} \end{cases}

[−−→ (x : τ) → s]_I = (x : [τ] → s)

[−−→ x : c]_I = x [c]

[−−→ (y : τ) → τ]_c = (y : [τ] → [τ]_c)

[−−→ (y : τ) → x : c]_c = ((y : [τ] → x [c]) → [τ]_c)

[Ind(x : τ)(x)] = \begin{cases} \text{λ} x_1 : τ. \text{Erase}((\text{Ind}(x : [τ]_I)([τ]_c)_I x_I)) & \text{if needErase}(\text{Ind}(x : τ)(x)) \\ ([\text{Ind}(x : [τ]_I)([τ]_c)_I]), & \text{if in } \text{Prop} \\ \text{isprop } (\text{Ind}(x : [τ]_I)([τ]_c)_I)) & \text{isprop } (\text{Ind}(x : [τ]_I)([τ]_c)_I)) \\ \text{Ind}(x : [τ]_I)([τ]_c)_I) & \text{otherwise} \end{cases}

[Con(τ, n)] = \begin{cases} \text{λ} x_1 : τ. \text{Con}([τ], n) x_1 & \text{if needErase}(τ) \\ \text{Con}([τ], n) & \text{otherwise} \end{cases}

6 Applicability

7 Related works and conclusion

[13] Uses a similar monad to represent impredicativity.


Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N° 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


Resizing Prop down to an axiom
