Is Impredicativity Implicitly Implicit?

Stefan Monnier
Université de Montréal - DIRO, Montréal, Canada
monnier@iro.umontreal.ca

Nathaniel Bos
McGill University - SOCS, Montréal, Canada
nathaniel.bos@mail.mcgill.ca

Abstract

Of all the threats to the consistency of a type system, such as side effects and recursion, impredicativity is arguably the least understood. In this paper, we try to investigate it using a kind of blackbox reverse-engineering approach to map the landscape. We look at it with a particular focus on its interaction with the notion of *implicit* arguments, also known as *erasable* arguments.

More specifically, we revisit several famous type systems believed to be consistent and which do include some form of impredicativity, and show that they can be refined to equivalent systems where impredicative quantification can be marked as erasable, in a stricter sense than the kind of proof irrelevance notion used for example for Prop terms in systems like Coq.

We hope these observations will lead to a better understanding of why and when impredicativity can be sound. As a first step in this direction, we discuss how these results suggest some extensions of existing systems where constraining impredicativity to erasable quantifications might help preserve consistency.

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

Keywords and phrases Impredicativity, Pure type systems, Inductive types, Erasable arguments, Proof irrelevance, Implicit arguments, Universe polymorphism

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 our definitions so as to prevent the diagonalization and self-references that lead to logical inconsistencies. 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. Some people reject impredicativity outright, and indeed systems like Agda [8] demonstrate that you can go a long way without impredicativity, yet, many popular systems, like Coq [18], do include some limited form of impredicativity. But those limits tend to feel somewhat ad-hoc, making the overall system more complex, with unsatisfying corner cases. For this reason we feel there is still a need to try and better understand what those limits to impredicativity should look like.

Let's disappoint the optimistic reader right away: we won't solve this problem. But during the design of our experimental language Typer [24], we noticed a property shared by several existing impredicative systems, that seemed to link impredicativity and erasability. Some mathematicians, such as Carnap [13], have argued that impredicative quantification might be acceptable as long as those arguments are not used in a, we shall say, "significant" way. So in a sense this article investigates whether erasability might be such a notion of "insignificance".

The two main instances of impredicativity in modern type theory are probably Coq's Prop universe, which is designed to be erasable, and the propositional resizing axiom [27] which allows the use of impredicativity for all *mere propositions*, i.e. types whose inhabitants are all provably equal and hence erasable. For this reason, it is no ground breaking revelation to claim that there is an affinity between impredicativity and erasability, yet it is still unclear to what extent the two belong together nor which particular form of erasability would be the true soulmate of impredicativity.

While Coq and the propositional resizing axiom basically link impredicativity to the concept of erasure usually called *proof irrelevance*, where an argument is deemed erasable if its type has at most one inhabitant, in this article we investigate its connection to a different form of erasability, where an argument is deemed erasable if the function only uses it in type annotations. This is the notion of erasability found in systems like ICC* and EPTS [5, 22].

More specifically, in Section 3, we take various well-known impredicative systems, refine them with annotations of *erasability*, and then show that all impredicatively quantified arguments can be annotated as erasable. In other words, we show that those existing systems already *implicitly* restrict the arguments to their impredicative quantifications to be erasable. This suggests that maybe a good rule of thumb to keep impredicative quantification sound is to make sure its argument is always erasable.

Armed with this proverbial hammer, we then look at the two main limitations of impredicative quantification in existing systems: the restriction we call no-SELIT (which disallows strong elimination of large inductive types) in systems like Coq, and the fact that only the bottom universe can be impredicative. We then propose systems that replace those somewhat ad-hoc restrictions with the arguably less ad-hoc restriction that impredicative quantification is restricted to erasable quantification. The contributions of this work are:

- A proof that in CCω all arguments to impredicative functions are erasable.
- A proof that in the CIC resulting from extending CCω with inductive types in the impredicative universe, all arguments to impredicative functions and all *large* fields of inductive types are also erasable.
- A new calculus ECIC which lifts the no-SELIT restriction, i.e. it extends CIC with strong elimination of large inductive types.
- A proof that restricting impredicativity to erasable quantifiers does not directly make impredicativity in more than one universe consistent.
- A new calculus EpCCω with an impredicative universe polymorphism which allows more powerful forms of impredicativity, such as a Church encoding with strong elimination.
- As needed for some of the above contributions, we sketch an extension of ICC* with both inductive types. While this is straightforward, we do not know of such a system published so far, the closest we found being the one by Bernardo in [6] and Tejiscak's thesis [26].

$$
\begin{array}{llll}
\textit{(var)} & x, y, t, l & \in & \mathcal{V} \\
\textit{(sort)} & s & \in & \mathcal{S} \\
\textit{(argkind)} & k, c & ::= & \mathsf{n} \mid \mathsf{e} \\
\textit{(term)} & e, \tau & ::= & s \mid x \mid (x{:}\tau_1) \xrightarrow{k} \tau_2 \mid \lambda x{:}\tau \xrightarrow{k} e \mid e_1 @^k e_2 \\
\textit{(context)} & \Gamma & ::= & \bullet \mid \Gamma, x{:}\tau
\end{array}
$$

$$
\textit{primitive reductions:} \qquad (\lambda x{:}\tau \xrightarrow{k} e_1) @^k e_2 \quad \rightsquigarrow \quad e_1[e_2/x]
$$

Figure 1 Syntax and reduction rules of EPTS.

2 Background

Here we present the notion of erasability we use in the rest of the paper.

2.1 Erasable Pure Type Systems

The calculi we use in this paper are erasable pure type systems (EPTS) [22], which are pure type systems (PTS) [4] extended with a notion of erasability. We use a notation that makes it more clear that the erasability is just an annotation like that of colored pure type systems (CPTS) [7] where the color indicates which arguments are 'n'ormal and which are 'e'rasable. The syntax of the terms and computation rules are shown in Figure 1.

A specific EPTS is then defined by providing the triplet $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ which defines respectively the sorts, axioms, and rules of this system. The difference with a plain pure type system, is that the annotation on a function or function call has to match the annotation of the function's type and that the elements of $\mathcal{R}$ have an additional k indicating to which color this rule applies: rules in $\mathcal{R}$ have the form (k, s_1, s_2, s_3) which means that a function of color k taking arguments in universe s_1 to values in universe s_2 itself lives in universe s_3. For example, we can define an EPTS which defines a version of System-F with erasability as follows:

$$
\begin{array}{lcl}
\mathcal{S} & = & \{\ *,\ \Box\ \} \\
\mathcal{A} & = & \{\ (*,\ \Box)\ \} \\
\mathcal{R} & = & \{\ (k, *, *, *),\ (k, \Box, *, *) \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \}
\end{array}
$$

This version has 4 different abstractions, allowing both System-F's value abstractions λ and type abstractions Λ to be annotated as either erasable or normal. It is well known that System-F enjoys the phase distinction [9], which means that all types can be erased before evaluating the terms, so we could also define an EPTS equivalent to System-F with only 2 abstractions, using the following rules instead:

$$
\mathcal{R} = \{\ (\mathsf{n}, *, *, *),\ (\mathsf{e}, \Box, *, *)\ \}
$$

This is an example of an impredicative calculus where we can make all impredicative abstractions (in this case, those introduced by the rule $(\Box, *, *)$ in the PTS) erasable.

Figure 2 shows the typing rules of our EPTS. Compared to a normal CPTS, the only difference is that the typing rule for functions is split into N-Lam and E-Lam where E-Lam includes the additional constraint $x \notin \mathsf{fv}(e*)$ that enforces the erasability of the argument. The expression

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau}\ (\textsc{Var}) \qquad \frac{(s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 \ : \ s_2}\ (\textsc{Sort}) \qquad \frac{\Gamma \vdash e \ : \ \tau_1 \qquad \Gamma \vdash \tau_2 \ : \ s \qquad \tau_1 \simeq \tau_2}{\Gamma \vdash e \ : \ \tau_2}\ (\textsc{Conv})$$

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

$$\frac{\Gamma \vdash e_1 \ : \ (x{:}\tau_1) \xrightarrow{k} \tau_2 \qquad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash e_1 @^k e_2 \ : \ \tau_2[e_2/x]}\ (\textsc{App}) \qquad \frac{\Gamma \vdash \tau_1 \ : \ s \qquad \Gamma, x{:}\tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 \xrightarrow{\mathsf{n}} e \ : \ (x{:}\tau_1) \xrightarrow{\mathsf{n}} \tau_2}\ (\textsc{n-Lam})$$

$$\frac{\Gamma \vdash \tau_1 \ : \ s \qquad \Gamma, x{:}\tau_1 \vdash e \ : \ \tau_2 \qquad x \notin \mathsf{fv}(e*)}{\Gamma \vdash \lambda x{:}\tau_1 \xrightarrow{\mathsf{e}} e \ : \ (x{:}\tau_1) \xrightarrow{\mathsf{e}} \tau_2}\ (\textsc{e-Lam})$$

Figure 2 Typing rules of our EPTS.
In the Conv rule, $\simeq$ stands for the ordinary β-convertibility.

"$e*$" is the *erasure* of e, where the erasure function $(\cdot)*$ erases type annotations as well as all erasable arguments:

$$\begin{array}{lcl}
s* & = & s \\
x* & = & x \\
((x{:}\tau_1) \xrightarrow{k} \tau_2)* & = & (x{:}\tau_1*) \to \tau_2* \\
(\lambda x{:}\tau \xrightarrow{\mathsf{n}} e)* & = & \lambda x \to e* \\
(\lambda x{:}\tau \xrightarrow{\mathsf{e}} e)* & = & e* \\
(e_1 @^{\mathsf{n}} e_2)* & = & e_1* @ e_2* \\
(e_1 @^{\mathsf{e}} e_2)* & = & e_1*
\end{array}$$

This expresses the fact that erasable arguments do not influence evaluation. The codomain of the erasure function is technically another language with a slightly different syntax, i.e. without erasability nor type annotations, but we will gloss over those details here since for the purpose of this article we only really ever need to know if "$x \in \mathsf{fv}(e*)$" rather than the specific shape of "$e*$" itself.

Since the new e-Lam rule is strictly more restrictive than the normal one, it is trivial to show that every EPTS S, just like every CPTS, has a corresponding PTS we note $\lfloor S \rfloor$ where erasability annotations have simply be removed, and that any well-typed term e in the EPTS S has a corresponding well-typed term $\lfloor e \rfloor$ in $\lfloor S \rfloor$. More specifically: $\Gamma \vdash e \ : \ \tau$ in the EPTS S implies $\lfloor \Gamma \rfloor \vdash \lfloor e \rfloor \ : \ \lfloor \tau \rfloor$ in the PTS $\lfloor S \rfloor$. As a corollary, if the corresponding PTS is consistent, the EPTS is also consistent.

2.2 Kinds of erasability

The claim that arguments to impredicative functions can be erased could be considered as trivial if we consider that Coq's only impredicative universe is Prop and that it is also the universe that gets erased during program extraction.

But the kind of erasability we use in this article is different from that offered by Coq's irrelevance of Prop: on the one hand it's more restrictive since the only thing you can do with an erasable argument in an EPTS is to pass it around until you finally put it inside a

$$
\begin{array}{rcll}
\mathcal{S} & = & \{\ \mathsf{Prop}; \mathsf{Type}_\ell \mid \ell \in \mathbb{N}\ \} & \\
\mathcal{A} & = & \{\ (\mathsf{Prop} : \mathsf{Type}_0); (\mathsf{Type}_\ell : \mathsf{Type}_{\ell+1}) & \mid\ \ell \in \mathbb{N}\ \} \\
\mathcal{R} & = & \{\ (k, \mathsf{Prop}, s, s) & \mid\ k \in \{\mathsf{n}, \mathsf{e}\}, s \in \mathcal{S}\ \} \\
 & & \cup\ \{\ (k, \mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ k \in \{\mathsf{n}, \mathsf{e}\}, \ell_1, \ell_2 \in \mathbb{N}\ \} \\
 & & \cup\ \{\ (\mathsf{e}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop}) & \mid\ \ell \in \mathbb{N}\ \} \\
 & & \cup\ \{\ (\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop}) & \mid\ \ell \in \mathbb{N}\ \} \Leftarrow \text{Rule absent from eCC}\omega \text{ and eCIC}
\end{array}
$$

Figure 3 Definition of CCω (and its little sibling eCCω) as EPTS.

type annotation, but on the other it's more flexible because any argument can be erasable, regardless of its type. For example, let us take the following polymorphic identity function in Coq:

```
Definition identity (t : Prop) (x : t) := x.
```

We can see that this function is impredicative since "`t`" can be instantiated with the type of `identity`. Coq's erasure would erase all uses of this function in terms that do not live in Prop, whereas we will concentrate here on the fact that the "`t`" argument is erasable because it is only used in type annotations.

In [2], Abel and Scherer discuss various other subtly different notions of erasure. One of the differences they mention is the difference between internal and external erasure. The rules of our EPTS are different in this respect from those of ICC [21] and ICC*[5]: our Conv rule requires convertibility of the fully explicit types (which corresponds to external erasure), whereas ICC and ICC* use a rule where convertibility is checked after erasure (so-called internal erasure):

$$\frac{\Gamma \vdash e\ :\ \tau_1 \qquad \Gamma \vdash \tau_2\ :\ s \qquad \tau_1 * \simeq \tau_2 *}{\Gamma \vdash e\ :\ \tau_2}$$

We use the weaker rule because it is sufficient for our needs and makes it immediately obvious that every well-typed term e in an EPTS S has a corresponding well-typed term $\lfloor e \rfloor$ in $\lfloor S \rfloor$. Our results would carry over to systems with the stronger rule, of course.

3 Erasable impredicativity in Prop

In this section we show that the impredicative quantification in the bottom universe Prop is almost always erasable and armed with this observation along with some circumstantial evidence, we propose to rely on this property in order to lift the no-SELIT restriction.

3.1 eCCω: Erasing impredicative arguments of CCω

We will start by showing that impredicative arguments in the calculus of constructions extended with a tower of universes (CCω) are always erasable. We use CCω, shown in Figure 3, because it is arguably the pure type system that is most closely related to existing systems like Coq. It follows the tradition of having a special impredicative Prop universe with a tower of predicative universes named Type_ℓ. $\mathsf{max}(\ell_1, \ell_2)$ denotes simply the least upper bound of l_1 and l_2.

The calculus $\lfloor \text{CC}\omega \rfloor$ we get by removing the erasability annotations is sometimes also called CCω in the literature. And indeed the two are equivalent: we can see that any well-typed term e in $\lfloor \text{CC}\omega \rfloor$ has a corresponding well-typed term $\lceil e \rceil$ in CCω such that $\lfloor \lceil e \rceil \rfloor = e$ by simply making $\lceil \cdot \rceil$ add n annotations everywhere. Our calculus CCω is incidentally almost identical to the ICC* calculus of Barras and Bernardo [5] (except for the Conv rule, as discussed above).

With respect to impredicativity, the relevant rules in CCω are $(\mathsf{e}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ and $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ which allow functions in Prop to take arguments in any Type_ℓ. We will now show that the second rule is redundant:

▶ **Lemma 1** (Confinement of impredicativity in CCω).
In CCω, if $\Gamma \vdash x \ : \ \tau_x$ and $\Gamma \vdash e \ : \ \tau_e$ and $\Gamma \vdash \tau_x \ : \ \mathsf{Type}_\ell$ and $\Gamma \vdash \tau_e \ : \ \mathsf{Prop}$ then x can only appear in $e*$ within arguments to impredicative functions, i.e. functions whose return values live in Prop and whose arguments don't.

Proof. By induction on the type derivation of e:

- Given $\tau_e : \mathsf{Prop}$, clearly e is too small to be a type like a sort s or an arrow $(y : \tau_1) \xrightarrow{k} \tau_2$, and it is also too small to be x itself.
- If the derivation uses the Conv rule to convert $e : \tau_e$ to $e : \tau'_e$, we know that τ'_e also has type Prop, by virtue of the type preservation property, so we can use the induction hypothesis on $e : \tau'_e$.
- If e is a function $\lambda y : \tau_y \xrightarrow{k} e_y$, then τ_y does not matter since it is erased from $e*$ and only occurrences of x in e_y is a concern, and since $\tau_e : \mathsf{Prop}$, we also know that the type of e_y is itself in Prop, hence we can use the induction hypothesis on it.
- If e is an application $e_1 @^k e_2$, as above we can apply the induction hypothesis to e_1. As for e_2, there are two cases: either e_1 takes an argument of type $\tau_1 : \mathsf{Prop}$ in which case we can again apply the induction hypothesis, or it takes an argument of type $\tau_1 : \mathsf{Type}_{\ell'}$ in which case we're done. ◀

We call eCCω the restriction of CCω where all arguments to impredicative functions are erasable, i.e. $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ is removed, as shown in Figure 3.

▶ **Theorem 2** (Erasability of impredicative arguments in CCω).
CCω's rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ is redundant, that is, for any derivation $\Gamma \vdash e \ : \ \tau$ in CCω there is a corresponding derivation $\Gamma' \vdash e' \ : \ \tau'$ in eCCω such that $\lfloor \Gamma \vdash e \ : \ \tau \rfloor = \lfloor \Gamma' \vdash e' \ : \ \tau' \rfloor$.

Proof. By induction on the type derivation of e where we systematically replace n with e on all functions, arrows, and applications that previously relied on the rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$. Since the erasability annotation is only used in the typing rule of λ-abstractions, the proof follows trivially for all cases except this one. For λ-abstractions that had an n annotation that we need to convert to e, we need to satisfy the additional condition that $x \notin \mathsf{fv}(e*)$, which follows from Lemma 1: In the absence of the rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$, all functions of type $(y : \tau_1) \xrightarrow{k} \tau_2$ where $\tau_2 : \mathsf{Prop}$ and $\tau_1 : \mathsf{Type}_{\ell'}$ are necessarily erasable, so Lemma 1 implies that x can never occur in $e'*$. ◀

$$
\begin{array}{llcl}
\textit{(index)} & i & \in & \mathbb{N} \\
\textit{(term)} & e, \tau, a, b, p & ::= & \ldots \mid \mathsf{Ind}(x{:}\tau)\langle \vec{a} \rangle \\
 & & & \mid \mathsf{Con}(i, \tau) \\
 & & & \mid \langle \tau_r \rangle \mathsf{Case}\ e\ \mathsf{of}\ \langle \vec{b} \rangle \\
 & & & \mid \mathsf{Fix}_i\ x : \tau\ =\ e
\end{array}
$$

$$
\begin{array}{lll}
\textit{primitive reductions:} & \langle \tau_r \rangle \mathsf{Case}\ (\mathsf{Con}(i, \tau)\ \overrightarrow{@^k e})\ \mathsf{of}\ \langle \vec{b} \rangle & \rightsquigarrow\ b_i\ \overrightarrow{@^k e} \\
 & \mathsf{Fix}_i\ x : \tau\ =\ e & \rightsquigarrow\ e[(\mathsf{Fix}_i\ x : \tau\ =\ e)/x]
\end{array}
$$

Figure 4 Extension of Figure 1's EPTS with inductive types.

This shows that the erasability of System-F's impredicative type abstractions can be extended to all of CCω's impredicative abstractions as well.

3.2 eCIC: Erasing impredicative arguments of CIC

We now extend this result to a calculus of inductive constructions (CIC). We reuse CCω as the base language and add inductive types to it. The term CIC has been used to refer to many different systems. Here we use it to refer to a variant of the "original" CIC from 1994, which only had 3 universes, in which we collapsed Set and Prop into a single universe, which we call Prop even though it is not restricted to be proof irrelevant like Coq's Prop; for readers more familiar with Coq, our CIC's Prop is more like Coq's impredicative Set. Note also that our CIC does have a tower of universes, like Coq, but its inductive types only exist in the bottom universe, as was the case in the original CIC, which is why we prefer to call it CIC than CICω.

We mostly follow the presentation of Giménez [16] for the syntax of inductive types but we extend its rules according to the presentation of Werner [29] which adds a strong elimination, i.e. the ability to compute a type by case analysis on an inductive type, which is needed for many proofs, even simple ones. The syntax of terms and the computational rules of inductive types are shown in Figure 4. Together with the rules of Figure 3 they define CIC (and its little sibling eCIC).

$\mathsf{Ind}(x{:}\tau)\langle \vec{a} \rangle$ is a (potentially indexed) inductive type which itself has type τ and whose i^{th} constructor has type a_i, where we use the vector notation $\vec{a}$ to represent a sequence of terms $a_0 \ldots a_n$. $\mathsf{Con}(i, \tau)$ denotes the i^{th} constructor of the inductive type e. $\langle \tau_r \rangle \mathsf{Case}\ e\ \mathsf{of}\ \langle \vec{b} \rangle$ is a case analysis of the term e which should be an object of inductive type; it will dispatch to the corresponding branch b_i if e was built with the i^{th} constructor of the inductive type; τ_r describes the return type of the case expression. Finally $\mathsf{Fix}_i\ x : \tau\ =\ e$ is a recursive function x of type τ, defined by structural induction on its i^{th} argument (the reduction rule shown above is naive, but the details do not affect us here).

We must of course also extend the definition of our erasure function to handle those additional terms:

$$
\begin{array}{lcl}
\mathsf{Ind}(x{:}\tau)\langle \vec{a} \rangle * & = & \mathsf{Ind}(x)\langle \overrightarrow{a*} \rangle \\
\mathsf{Con}(i, \tau) * & = & \mathsf{Con}(i) \\
\langle \tau_r \rangle \mathsf{Case}\ e\ \mathsf{of}\ \langle \vec{b} \rangle * & = & \mathsf{Case}\ e*\ \mathsf{of}\ \langle \overrightarrow{b*} \rangle \\
(\mathsf{Fix}_i\ x : \tau\ =\ e) * & = & \mathsf{Fix}\ x\ =\ e*
\end{array}
$$

$$\frac{\Gamma \vdash \tau \ : \ s \qquad \forall i. \qquad \Gamma, x{:}\tau \vdash a_i \ : \ \mathsf{Prop} \qquad x \vdash a_i \ \ \mathsf{con}}{\Gamma \vdash \mathsf{Ind}(x{:}\tau)\langle \vec{a} \rangle \ : \ \tau}$$

$$\frac{\tau = \mathsf{Ind}(x{:}\tau')\langle \vec{a} \rangle \qquad \Gamma \vdash \tau \ : \ \tau'}{\Gamma \vdash \mathsf{Con}(i, \tau) \ : \ a_i[\tau/x]} \qquad \frac{\forall i. \qquad \Gamma \vdash \tau_i \ : \ \mathsf{Prop}}{\Gamma \vdash \vec{\tau} \ \ \mathsf{small}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \ : \ \tau_I \ \overrightarrow{@^k p} \qquad \tau_I = \mathsf{Ind}(\overrightarrow{x{:}(z{:}\tau_z) \xrightarrow{k}} \mathsf{Prop})\langle \vec{a} \rangle \qquad \Gamma \vdash \tau_r \ : \ \overrightarrow{(z{:}\tau_z) \xrightarrow{k}} (\_{:}\tau_I \ \overrightarrow{@^k z}) \xrightarrow{\mathsf{n}} s \\ \forall i. \qquad a_i = \overrightarrow{(y{:}\tau_y) \xrightarrow{c}} x \ \overrightarrow{@^k p'} \qquad s = \mathsf{Prop} \vee \Gamma \vdash \vec{\tau_y} \ \ \mathsf{small} \\ \forall i. \qquad \Gamma \vdash b_i \ : \ \overrightarrow{(y{:}\tau_y[\tau_I/x]) \xrightarrow{c}} (\tau_r \ \overrightarrow{@^k p'} @^{\mathsf{n}}(\mathsf{Con}(i, \tau_I) \ \overrightarrow{@^c y}))\end{array}}{\Gamma \vdash \langle \tau_r \rangle \mathsf{Case} \ e \ \mathsf{of} \ \langle \vec{b} \rangle \ : \ \tau_r \ \overrightarrow{@^k p} @^{\mathsf{n}} e}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau \ : \ s \\ \Gamma, x_f{:}\tau \vdash e \ : \ \tau \qquad e = \overrightarrow{\lambda y{:}\_ \xrightarrow{k}} \lambda x_i{:}\_ \xrightarrow{k} e_b \qquad i = |y| \qquad x_f; i; x_i; \emptyset \vdash e_b \ \ \mathsf{term}\end{array}}{\Gamma \vdash \mathsf{Fix}_i \ x_f : \tau \ = \ e \ : \ \tau}$$

Figure 5 Typing rules of inductive types.

Auxiliary judgments: $\Gamma \vdash \vec{\tau} \ \ \mathsf{small}$ checks that the fields $\vec{\tau}$ are all in Prop.
$x \vdash a_i \ \ \mathsf{con}$ checks that a is strictly positive in x.
$x_f; i; x_i; \emptyset \vdash e_b \ \ \mathsf{term}$ makes sure all recursive calls use structurally decreasing arguments.

While these new terms may appear not to take erasability into account, this is only because the erasability of the fields of those inductive types is introduced by the erasability annotations on the formal arguments of $\vec{a}$ which need to match those of $\vec{b}$: they really do let you specify the erasability of each field; and every field, whether erasable or not, is available within the corresponding Case branch but those marked as erasable in the Ind definition will accordingly only be available as erasable within Case.

Figure 5 shows the typing rules corresponding to each of those four new constructs. Those typing rules are pretty intricate, if not downright scary, and most of the details do not directly affect our argument, so the casual reader may prefer to skip them. We use _ at a few places where the actual element does not matter enough to give it a name. The notation $f \ \overrightarrow{@^k e}$ denotes a curried application with multiple arguments $f @^{k_1} e_1 \ldots @^{k_n} e_n$, and similarly $\overrightarrow{\lambda x{:}\tau \xrightarrow{k}} e$ denotes a curried function of multiple arguments $\lambda x_1{:}\tau_1 \xrightarrow{k_1} \ldots \lambda x_n{:}\tau_n \xrightarrow{k_n} e$ and $\overrightarrow{(x{:}\tau) \xrightarrow{k}} e$ denotes the type of such a function $(x_1{:}\tau_1) \xrightarrow{k_1} \ldots (x_n{:}\tau_n) \xrightarrow{k_n} e$.

The rules are very similar to those used by Giménez in [16] because they are largely unaffected by the erasability annotations. The only exception is for Case where we have to make sure that the various erasability annotations match each other, e.g. the vector $\vec{c}$ of erasability annotations placed on a given constructor a_i must match the erasability annotations of the arguments expected by the corresponding branch b_i. Two important details are worth pointing out:

- In the rule for Ind the type of constructors is restricted to be in Prop: just like in the original CIC we only allow inductive types in our bottom universe, contrary to what

systems like Coq [18] and UTT [20] allow.

- In the Case rule, the hypotheses $s = \mathsf{Prop} \vee \Gamma \vdash \vec{\tau_y}$ small ensure that when the result of the case analysis is not in Prop, i.e. when this is a form of strong elimination, the inductive type must be small, meaning that all its fields must be in Prop. This "no-SELIT" restriction is taken from Werner [29], with a slightly different presentation because he chose to split the Case rule into two: one for weak elimination and one for strong elimination.

We do not show the definition of the $x \vdash e$ con judgment which ensures that e has the appropriate shape for an inductive constructor, including the strict positivity, nor that of the $x_f; i; x_i; \nu \vdash e$ term judgment which ensures that recursive calls are made on structurally smaller terms. Their definition is not affected by the presence of erasability annotations and does not impact our work here.

To show that the $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ rule of non-erasable impredicativity is still redundant in this new system, we proceed in the same way:

▶ **Lemma 3** (Confinement of impredicativity in CIC)**.**
In CIC, if $\Gamma \vdash x : \tau_x$ and $\Gamma \vdash e : \tau_e$ and $\Gamma \vdash \tau_x : \mathsf{Type}_\ell$ and $\Gamma \vdash \tau_e : \mathsf{Prop}$ then x can only appear in $e*$ within arguments to impredicative functions, i.e. functions whose return values live in Prop and whose arguments don't.

Proof. The proof stays the same as for CCω, with the following additional cases:

- Given $\tau_e : \mathsf{Prop}$, clearly e is too small to be a type like $\mathsf{Ind}(x{:}\tau)\langle\vec{a}\rangle$.
- If e is of the form $\mathsf{Con}(i, \tau)$, since τ is erased, the erasure is always closed.
- If e is of the form $\mathsf{Fix}_i\ x : \tau = e'$, then τ does not matter because it's erased, and we can invoke the inductive hypothesis on e'.
- If e is of the form $\langle\tau_r\rangle\mathsf{Case}\ e'\ \mathsf{of}\ \langle\vec{b}\rangle$, then τ_r does not matter because it is erased. Furthermore, we can invoke the inductive hypothesis on e' since we know that e' lives in Prop, like all our inductive types. Finally since the hypothesis tells us that e lives in Prop, all branches b_i must as well, hence we can also invoke the induction hypothesis on every b_i. ◀

We call eCIC the restriction of CIC where all arguments to impredicative functions and all large fields of inductive definitions are erasable, i.e. $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ is removed.

▶ **Theorem 4** (Erasability of impredicative arguments in CIC)**.**
CIC's rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ is redundant, that is, for any derivation $\Gamma \vdash e : \tau$ in CIC there is a corresponding derivation $\Gamma' \vdash e' : \tau'$ in eCIC such that $\lfloor \Gamma \vdash e : \tau \rfloor = \lfloor \Gamma' \vdash e' : \tau' \rfloor$

Proof. As before, by induction on the type derivation of e where we systematically replace n with e on all functions, arrows, and applications that previously relied on the rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$. The interesting new case is when e is of the form $\langle\tau_r\rangle\mathsf{Case}\ e'\ \mathsf{of}\ \langle\vec{b}\rangle$: as mentioned, the vector $\vec{c}$ of erasability annotations placed on a given constructor a_i must match the erasability annotations of the arguments expected by the corresponding branch b_i. Since our inductive types all live in Prop, it means all fields that live in higher universes have been annotated as erasable. But that in turns means that all corresponding arguments to the branches b_i should also be annotated as erasable. When s is Prop (i.e. a weak elimination), this is the case because all arguments of higher universe for functions in Prop can only be

$$\begin{array}{rcll}\mathcal{R} & = & \{\ (k, \mathsf{Prop}, s, s) & \mid\ k \in \{\mathsf{n}, \mathsf{e}\}, s \in \mathcal{S}\ \} \\ & \cup & \{\ (k, \mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ k \in \{\mathsf{n}, \mathsf{e}\}, \ell_1, \ell_2 \in \mathbb{N}\ \} \\ & \cup & \{\ (\mathsf{e}, \mathsf{Type}_{\ell}, \mathsf{Prop}, \mathsf{Prop}) & \mid\ \ell \in \mathbb{N}\ \}\end{array}$$

$$\frac{\Gamma \vdash e\ :\ \tau_I\ \overrightarrow{@^k p} \qquad \tau_I = \mathsf{Ind}(x : \overrightarrow{(z:\tau_z) \xrightarrow{k} \mathsf{Prop}})\langle\vec{a}\rangle \qquad \Gamma \vdash \tau_r\ :\ \overrightarrow{(z:\tau_z) \xrightarrow{k} (\_ : \tau_I\ \overrightarrow{@^k z}) \xrightarrow{\mathsf{n}} s} \\ \forall i. \quad a_i = \overrightarrow{(y:\tau_y) \xrightarrow{c} x\ \overrightarrow{@^k p'}} \qquad \Gamma \vdash b_i\ :\ \overrightarrow{(y : \tau_y[\tau_I/x]) \xrightarrow{c} (\tau_r\ \overrightarrow{@^k p'} @^{\mathsf{n}} (\mathsf{Con}(i, \tau_I)\ \overrightarrow{@^c y}))}}{\Gamma \vdash \langle\tau_r\rangle \mathsf{Case}\ e\ \mathsf{of}\ \langle\vec{b}\rangle\ :\ \tau_r\ \overrightarrow{@^k p}\, @^{\mathsf{n}} e}$$

Figure 6 Rules of the ECIC system. The rest is unchanged from eCIC, Figures 1, 2, 4, and 5.

annotated as erasable. And when s is a higher universe the property is also verified because the $\Gamma \vdash \vec{\tau_y}$ small constraint imposes that none of the arguments are in higher universes so they don't use the $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ rule. ◀

This shows that the erasability of System-F's impredicative type abstractions can be extended not only to all of CCω's impredicative abstractions but also to CIC's impredicative abstractions and impredicative inductive types.

3.3 ECIC: Strong elimination of large inductive types

The reason behind the $\Gamma \vdash e$ small special constraint on strong eliminations of CIC in Figure 5 is pretty straightforward: without this restriction, we could use an inductive type such as the following to "smuggle" a value of universe Type_ℓ in a box of universe Prop:

```
Inductive Box (t : Type): Prop := box : t -> Box.
Definition unbox (t : Type) (x : Box t) := match x with
  | box x' => x'
end.
```

Note that such a box (a large inductive type) is perfectly valid in CIC, but the $\Gamma \vdash e$ small constraint rejects the `unbox` definition (which uses a strong elimination). If we remove the $\Gamma \vdash e$ small constraint, the effect of such a `box`/`unbox` pair would be to lower any value of a higher universe to the Prop universe and would hence defeat the purpose of the stratification introduced by the tower of universes. This was first shown to be inconsistent in [11].

This restriction makes the system more complex since elimination is allowed from any inductive type to any universe except for the one special case of strong elimination of large inductive types (SELIT). It also significantly weakens the system. For example, in Coq with the `--impredicative-set` option, we can define a large inductive type like:

```
Inductive Ω : Set :=
  | int   : Ω
  | arrow : Ω -> Ω -> Ω
  | all   : forall k:Set, (k -> Ω) -> Ω.
```

which could be used for example to represent the types of some object language. But we cannot prove properties such as the following variant of Leibniz equality (which we needed in the proof of soundness of our Swiss coercion [23]):

```
forall k₁ k₂ f₁ f₂ p,
  all k₁ f₁ = all k₂ f₂  ->  p k₁ f₁ -> p k₂ f₂.
```

In practice, this important restriction significantly reduces the applicability of large inductive types (which partly explains why Coq does not allow them in Set any more by default).

While the $\Gamma \vdash e$ small constraint was added to avoid an inconsistency, this same $\Gamma \vdash e$ small is also the key to making our proof of erasability of impredicative arguments work for CIC: it is the detail which makes it possible to mark all the large fields of impredicative inductive definitions as erasable, as we saw in the previous section. This might be a coincidence, of course, yet it suggests a close alignment between the needs of consistency and the need to keep impredicative elements erasable.

Figure 6 shows a refinement of eCIC we call ECIC whose Case rule does not have the $\Gamma \vdash e$ small constraint. ECIC is more elegant and regular than CIC thanks to the absence of this special corner case, and it allows typing more terms than eCIC and hence CIC. For instance in ECIC we can define the above Ω inductive type with an erasable k and then prove the mentioned property (with k_1 and k_2 marked as erasable).

Note also that the lack of an $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ rule, means we cannot define a `box` as above in this system; instead we are limited to making its content erasable. This in turn prevents us from defining `unbox` since the `x'` would now be erasable so it cannot be returned as-is from the elimination form. In other words, forcing impredicative fields to be erasable also avoids this source of inconsistency usually avoided with the $\Gamma \vdash e$ small constraint. Based on this circumstantial evidence, we venture to state the following:

▶ **Conjecture 5.** *The ECIC system is consistent.*

3.4 SELIT for Coq's proof-irrelevant Prop

The Prop universe used in the previous section corresponds to Coq's impredicative Set universe, which is disabled by default. Coq's impredicative Prop universe is similar except it is designed to be proof-irrelevant. This property is used in two ways: to reflect this property in the system via an axiom and to erase all Prop terms when *extracting* a program from a proof. This proof-irrelevance property is enforced by two constraints imposed on the strong elimination of those inductive types that live in Prop: first, they have to have a single constructor and second, all fields must live in the Prop universe. The first constraint makes sure there is no run-time dispatch based on an erased value, while the second guarantees that the only data we can extract from an erased value is itself erased.

The second constraint is the no-SELIT constraint. So the Conjecture 5 suggests we could relax this restriction and allow strong elimination on any Prop type with a single constructor if the fields that do not live in Prop are erasable. From the point of view of extraction, we could even relax this further to allow strong elimination on any Prop type with a single constructor, and simply treat all the values so extracted as erased.

3.5 eCoq: Erasing impredicativity in Coq and UTT

As noted in Section 3.2, we were careful to restrict our inductive types to live in Prop. This was no accident: we rely on this property in the confinement lemma used to show the erasability of all impredicative arguments in CIC. Indeed, confinement does not hold if we can do a case analysis on an inductive type that lives in Type_ℓ and return a value in Prop.

$$
\begin{array}{rcll}
\mathcal{R} & = & \{\ (k, \mathsf{Prop}, s, s) & \mid\ k \in \{\mathsf{n}, \mathsf{e}\}, s \in \mathcal{S}\ \} \\
& \cup & \{\ (\mathsf{e}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop}) & \mid\ \ell \in \mathbb{N}\ \} \\
& \cup & \{\ (\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Type}_\ell) & \mid\ \ell \in \mathbb{N}\ \} \\
& \cup & \{\ (k, \mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\mathsf{max}(\ell_1,\ell_2)}) & \mid\ k \in \{\mathsf{n}, \mathsf{e}\}, \ell_1, \ell_2 \in \mathbb{N}\ \}
\end{array}
$$

$$
\frac{\Gamma \vdash \tau\ :\ s \qquad \forall i. \qquad \Gamma, x{:}\tau \vdash a_i\ :\ s' \qquad x \vdash a_i\ \mathsf{con}}{\Gamma \vdash \mathsf{Ind}(x{:}\tau)\langle \vec{a} \rangle\ :\ \tau}
$$

$$
\frac{\begin{array}{l}\Gamma \vdash e\ :\ \tau_I\ \overrightarrow{@^k p} \qquad \tau_I = \mathsf{Ind}(x{:}\overrightarrow{(z{:}\tau_z) \xrightarrow{k} s'})\langle \vec{a} \rangle \qquad \Gamma \vdash \tau_r\ :\ \overrightarrow{(z{:}\tau_z) \xrightarrow{k} (\_{:}\tau_I\ \overrightarrow{@^k z}) \xrightarrow{\mathsf{n}} s} \\ \forall i. \qquad a_i = \overrightarrow{(y{:}\tau_y) \xrightarrow{c} x\ \overrightarrow{@^k p'}} \qquad \Gamma \vdash b_i\ :\ \overrightarrow{(y{:}\tau_y[\tau_I/x]) \xrightarrow{c} (\tau_r\ \overrightarrow{@^k p'} @^{\mathsf{n}} (\mathsf{Con}(i, \tau_I)\ \overrightarrow{@^c y}))}\end{array}}{\Gamma \vdash \langle \tau_r \rangle \mathsf{Case}\ e\ \mathsf{of}\ \langle \vec{b} \rangle\ :\ \tau_r\ \overrightarrow{@^k p}\, @^{\mathsf{n}} e}
$$

Figure 7 Rules of the eCoq system.

Systems such as Coq and UTT [20] allow impredicative definitions in Prop, inductive types in higher universes, and elimination from those inductive types to Prop. These systems are hence examples of impredicativity which is not straightforwardly erasable like it is in the systems seen so far. Here is an example of code which relies on this possibility:

```
Inductive List (α : Type₀) : Type₀ := nil | cons (v : α) (vs : List t).

Definition ifnil (ts : List Prop) (t : Prop) (x y : t) :=
  match ts with
  | nil => x
  | cons _ _ => y.
```

In Coq, `ifnil` lives in Prop because its return value is in Prop. If we extend Coq with erasability annotations, the argument "`t`" could be marked as erasable since it only appears in type annotations, but not the other three arguments. To determine in which universe it rests, we would use the rules $(\mathsf{n}, \mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop})$ for the last two arguments and $(\mathsf{e}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ for the second argument. Those rules obey the principle that impredicativity is restricted to erasable arguments. But for the first argument, we need the rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ which does not obey this principle.

If we want to obey the principle, we could replace this last rule with the predicative rule $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Type}_\ell)$ instead. Figure 7 shows the important rules of such a system we call eCoq. With such a system, we would have to adjust the above example in one of two ways:

- Live with the fact that `ifnil` will now live in Type_0 rather than in Prop.
  Experience with Agda and other systems suggests that most code does not rely on impredicativity, so in practice this first approach should be applicable in most cases.
- Mark the non-Prop parts of "`ts`" as erasable so that it can live in Prop. Concretely, it means using a new type we could call `eList`, which is like `List` except that the "`v`" field of the "`cons`" constructor is marked as erasable, to allow those "thinner" lists to live in Prop.

We call the second approach *thinning*. It replaces inductive objects from a higher universe

$$
\begin{array}{rcl}
\mathcal{U} & = & \Pi\mathcal{X} : \Box.((\wp\wp\mathcal{X} \to \mathcal{X}) \to \wp\wp\mathcal{X}) \\
\tau t & = & \Lambda\mathcal{X} : \Box.\lambda f : (\wp\wp\mathcal{X} \to \mathcal{X}).\lambda p : \wp\mathcal{X}.(t\ \lambda x : \mathcal{U}.(p\ (f\ (\{x\ \mathcal{X}\}\ f)))) \\
\sigma s & = & (\{s\ \mathcal{U}\}\ \lambda t : \wp\wp\mathcal{U}.\tau t) \\
\Delta & = & \lambda y : \mathcal{U}.\neg\forall p : \wp\mathcal{U}.[(\sigma y\ p) \Rightarrow (p\ \tau\sigma y)] \\
\Omega & = & \tau\ \lambda p : \wp\mathcal{U}.\forall x : \mathcal{U}.[(\sigma x\ p) \Rightarrow (p\ x)]
\end{array}
$$

$$
\begin{array}{l}
[\ \mathsf{suppose}\ 0 : \forall p : \wp\mathcal{U}.[\forall x : \mathcal{U}.[(\sigma x\ p) \Rightarrow (p\ x)] \Rightarrow (p\ \Omega)]. \\
\quad [\ [\langle 0 \Delta\rangle\ \mathsf{let}\ x : \mathcal{U}. \\
\qquad\qquad \mathsf{suppose}\ 2 : (\sigma x\ \Delta). \\
\qquad\qquad \mathsf{suppose}\ 3 : (\forall p : \wp\mathcal{U}.[(\sigma x\ p) \Rightarrow (p\ \tau\sigma x)]). \\
\qquad\qquad [[\langle 3\ \Delta\rangle\ 2]\ \mathsf{let}\ p : \wp\mathcal{U}.\langle 3\ \lambda y : \mathcal{U}.(p\ \tau\sigma y)\rangle]] \\
\qquad \mathsf{let}\ p : \wp\mathcal{U}.\langle 0\ \lambda y : \mathcal{U}.(p\ \tau\sigma y)\rangle] \\
\quad \mathsf{let}\ p : \wp\mathcal{U}. \\
\quad \mathsf{suppose}\ 1 : \forall x : \mathcal{U}.[(\sigma x\ p) \Rightarrow (p\ x)]. \\
\quad [\langle 1\ \Omega\rangle\ \mathsf{let}\ x : \mathcal{U}.\langle 1\ \tau\sigma x\rangle]]
\end{array}
$$

Figure 8 Hurken's paradox.

with similar objects that fit in Prop by marking the non-Prop parts of it as erasable or by replacing them with similarly "thinned" elements.

It is still unclear whether any valid typing derivation in a system like Coq can have a corresponding typing derivation in eCoq, that is, whether we can do away with the $(\mathsf{n}, \mathsf{Type}_\ell, \mathsf{Prop}, \mathsf{Prop})$ rule because we can always change the source code as described above.

4 Universe-agnostic impredicativity

CCω accepts impredicative definitions only in the bottom universe, Prop, just like in most known consistent type systems that support impredicative definitions (one counter example being arguably the λPREDω^+ presented in [14]). This is a direct consequence of various paradoxes formalized in systems which allow impredicative definitions in more than one universe [17, 12, 19]. In this section we investigate the use of erasability constraints in order to lift this restriction and thus allow impredicative definitions in higher universes as well.

4.1 $\lambda\mathsf{e}U^-$: Erasing impredicative arguments in λU^-

The last two papers referenced above showed a paradox in the system λU^- which is F_ω extended with one extra rule. It can be defined as an EPTS as follows:

$$
\begin{array}{rcll}
\mathcal{S} & = & \{\ *,\ \Box,\ \Delta\ \} & \\
\mathcal{A} & = & \{\ (*,\ \Box), (\Box,\ \Delta)\ \} & \\
\mathcal{R} & = & \{\ (k,*,*,*),\ (k,\Box,*,*),\ (k,\Box,\Box,\Box),\ (k,\Delta,\Box,\Box) & \mid k \in \{\mathsf{n},\mathsf{e}\}\ \}
\end{array}
$$

Two of the four pairs of rules are impredicative: $(k,\Box,*,*)$ and $(k,\Delta,\Box,\Box)$. The first is generally considered harmless since $*$ is the bottom universe and hence corresponds to Prop in CCω. The new one is $(k,\Delta,\Box,\Box)$ which introduces impredicativity in the second universe, $\Box$. Following the same idea as in the previous section where we defined ECIC to rely on

erasability to avoid inconsistency, we could thus define a new $\lambda \mathsf{e}U^-$ calculus that only allows the use of impredicativity with erasable abstractions:

$$\mathcal{R} \quad = \quad \{\ (k, *, *, *),\ (\mathsf{e}, \Box, *, *),\ (k, \Box, \Box, \Box),\ (\mathsf{e}, \Delta, \Box, \Box) \qquad\qquad \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \}$$

Alas, this does not help:

▶ **Theorem 6.** *$\lambda \mathsf{e}U^-$ is not consistent.*

Proof. The proof is the same as the proof of inconsistency of λU^- shown by Hurkens in [19]. Figure 8 shows Hurken's original proof, using the same notation he used in his paper. To show that the proof also applies to $\lambda \mathsf{e}U^-$, we need to make sure that all impredicative abstractions can be annotated as erasable. For that, it suffices to know that the integers are variable names, the impredicative abstraction in $*$ is introduced by let, the corresponding application is denoted with $\langle e_1\ e_2 \rangle$, the impredicative abstraction in $\Box$ is introduced by Λ, and the corresponding application is denoted with $\{e_1\ e_2\}$: by inspection we can see that all the arguments introduced by impredicative abstractions are exclusively used either in type annotations or in arguments to other impredicative functions. ◀

This demonstrates that, even though the notion of erasability we use here has shown strong affinities with consistent uses of impredicativity, it is not in general sufficient to tame the excesses of impredicativity.

4.2 Inductive types: Impredicative and universe polymorphic?

While paradoxes like Hurkens's suggest that it is impossible to have impredicative definitions in more than one universe without losing consistency, inductive definitions suggest otherwise.

The traditional encoding of inductive types using Church's impredicative encoding looks like the following:

$$NatC = (a : \mathsf{Prop}) \to a \to (a \to a) \to a$$

But this is much more restrictive than the usual definition of Nat as a real inductive type. More specifically, when defined as an inductive type we get two extra features compared to the above Church encoding: the ability to do dependent elimination, and the ability to perform elimination to any universe rather than only to Prop. Let us focus on the second one. The following Church-like encoding would lift this restriction, allowing elimination to any universe:

$$NatL = (l : \mathsf{Level}) \to (a : \mathsf{Type}_l) \to a \to (a \to a) \to a$$

Such a definition is possible in systems like Agda which provide the necessary universe polymorphism (the l above is a universe-level variable), but this type $NatL$ is traditionally placed in a universe too high to be useful as an encoding of natural numbers.

We have not been able to find a concise description of the rules used in Agda, but a first approximation of its type system is described informally in Figure 9 where ω stands for the smallest infinite ordinal. According to those rules, Agda would place the above universe-polymorphic definition of $NatL$ squarely in the far away Type_ω universe. Yet everything that can be done with it can also be done with the real Nat inductive type, which lives in the much more palatable Type_0 universe, so it would arguably be safe to let $NatL$ live in Type_0

$$
\begin{array}{lcll}
\textit{(level)} & \ell & ::= & 0 \mid \mathsf{s}\ \ell \mid l \mid \ell_1 \sqcup \ell_2 \\
\\
\mathcal{S} & = & \{\ \mathsf{UL};\ \mathsf{Type}_\ell;\ \mathsf{Type}_\omega\ \} & \\
\mathcal{A} & = & \{\ (\mathsf{Level} : \mathsf{UL});\ (\mathsf{Type}_\ell : \mathsf{Type}_{(\mathsf{s}\ \ell)})\ \} & \\
\mathcal{R} & = & \{\ (k, \mathsf{UL}, \mathsf{Type}_\ell, \mathsf{Type}_\omega) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \} \\
 & & \cup \{\ (k, \mathsf{UL}, \mathsf{Type}_\omega, \mathsf{Type}_\omega) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \} \\
 & & \cup \{\ (k, \mathsf{Type}_\ell, \mathsf{Type}_\omega, \mathsf{Type}_\omega) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \} \\
 & & \cup \{\ (k, \mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\ell_1 \sqcup \ell_2}) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \}
\end{array}
$$

Figure 9 Informal rules of an Agda-like system.

(and thus make this definition impredicative). The same reasoning applies to the following type:

$$\mathit{ListType} = (l : \mathsf{Level}) \to (a : \mathsf{Type}_l) \to a \to (\mathsf{Type}_0 \to a \to a) \to a$$

So *ListType* should arguably live in Type_1 rather than in Type_ω since that is what happens when defined as a real inductive type. This would also make *ListType* impredicative but should not threaten consistency. This illustrates that every inductive type corresponds to an impredicative definition that could live in the same universe, making it clear that having impredicative definitions in multiple universe levels is not inherently incompatible with consistency.

Of course, this begs the question: what is it that makes it safe to let those definitions be treated as impredicative? What is special about them?

In the rest of this section we will consider one hypothesis, which is that the universe level parameter ℓ needs to be erasable. In practice the vast majority of universe polymorphism can be marked as erasable. Some simple counter examples are:

$$
\begin{array}{l}
\mathit{Set} = \lambda l : \mathsf{Level} \xrightarrow{\mathsf{n}} \mathsf{Type}_l \\
\mathit{ListType} = \lambda l_1 : \mathsf{Level} \xrightarrow{\mathsf{n}} (l_2 : \mathsf{Level}) \xrightarrow{\mathsf{e}} (a : \mathsf{Type}_{l_2}) \xrightarrow{\mathsf{e}} a \xrightarrow{\mathsf{n}} (\mathsf{Type}_{l_1} \xrightarrow{\mathsf{n}} a \xrightarrow{\mathsf{n}} a) \xrightarrow{\mathsf{n}} a
\end{array}
$$

4.3 EpCCω: Impredicative erasable universe polymorphism

With universe polymorphism, sorts are not closed any more, so we cannot really represent the rules that govern them using a simple set like $\mathcal{R}$. So, the $(k, \mathsf{UL}, \mathsf{Type}_\ell, \mathsf{Type}_\omega)$ rule was really meant to say something like:

$$\frac{\Gamma \vdash \tau_1 \ :\ \mathsf{UL} \qquad \Gamma, l : \tau_1 \vdash \tau_2 \ :\ \mathsf{Type}_\ell}{\Gamma \vdash (l : \tau_1) \xrightarrow{k} \tau_2 \ :\ \mathsf{Type}_\omega}$$

Now if we want to make this impredicative when $k = \mathsf{e}$, since ℓ can refer to l we need to substitute l with *something* before we can use it in the sort of the product. For the *NatL* case, for example, ℓ will be "$\mathsf{s}\ l$" and we argued that this product type should live in Type_0, so we would need to substitute l with -1! Rather than argue why a negative value could make sense, we will use 0 in our rule:

$$\frac{\Gamma \vdash \tau_1 \ :\ \mathsf{UL} \qquad \Gamma, l : \tau_1 \vdash \tau_2 \ :\ \mathsf{Type}_\ell}{\Gamma \vdash (l : \tau_1) \xrightarrow{\mathsf{e}} \tau_2 \ :\ \mathsf{Type}_{\ell[0/l]}}$$

$$
\begin{array}{rcll}
\mathcal{R} & = & \{\ (\mathsf{n}, l:\mathsf{UL}, \mathsf{Type}_\ell, \mathsf{Type}_\omega)\ \} & \\
& \cup & \{\ (\mathsf{e}, l:\mathsf{UL}, \mathsf{Type}_\ell, \mathsf{Type}_{\ell[0/l]})\ \} & \\
& \cup & \{\ (k, l:\mathsf{UL}, \mathsf{Type}_\omega, \mathsf{Type}_\omega) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \} \\
& \cup & \{\ (k, t:\mathsf{Type}_\ell, \mathsf{Type}_\omega, \mathsf{Type}_\omega) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \} \\
& \cup & \{\ (k, t:\mathsf{Type}_{\ell_1}, \mathsf{Type}_{\ell_2}, \mathsf{Type}_{\ell_1 \sqcup \ell_2}) & \mid k \in \{\mathsf{n}, \mathsf{e}\}\ \}
\end{array}
$$

Figure 10 Informal rules of EpCCω.

While this places *NatL* in Type_1 rather than Type_0, it still makes it impredicative, and if all our base types live in Type_1 we will not notice much difference.

Figure 10 describes the resulting calculus we call EpCCω, where the second fields of elements of $\mathcal{R}$ now have the shape "$x : s$" so we can refer to the variable x that can appear freely in the third field.

4.4 Encoding System-F in EpCCω

EpCCω is basically a predicative version of CCω (hence the "p") to which we added universe polymorphism and impredicative erasable universe polymorphism (which motivated the "E"). Contrary to the previous calculus it does not have a base impredicative universe Prop: its only source of impredicativity is the $(\mathsf{e}, l : \mathsf{UL}, \mathsf{Type}_\ell, \mathsf{Type}_{\ell[0/l]})$ rule which introduces the impredicative erasable universe polymorphism. Compared to Agda, it lacks inductive types but it adds a form of impredicativity. While we do not know if it is consistent, we can try and compare it to existing systems, and for that we start by showing how to encode System-F. In order for our encoding function $[\![\cdot]\!]$ to be based purely on the syntax of terms rather than the typing derivation, we take as input a stratified version of System-F:

$$
\begin{array}{llcl}
\textit{(types)} & \tau & ::= & t \mid \tau_1 \to \tau_2 \mid (t:*) \to \tau \\
\textit{(terms)} & e & ::= & x \mid \lambda x{:}\tau \to e \mid e_1\ e_2 \mid \lambda t{:}* \to e \mid e\ \tau
\end{array}
$$

To encode System-F, the only interesting part is the need to simulate System-F's impredicative quantification over types. We can do that in the same way *NatC* was generalized to *NatL*, i.e. by replacing "$(t : *) \to \tau$" with "$(l:\mathsf{Level}) \xrightarrow{\mathsf{e}} (t:\mathsf{Type}_l) \xrightarrow{\mathsf{n}} \tau$". The only tricky aspect of this is that while in System-F all the type variables (and more generally all the types) have the same kind $*$, this encoding makes every type variable come with its own universe level, so the encoding function needs to keep track of the level of each type in order to know how to instantiate the $(l:\mathsf{Level}) \xrightarrow{\mathsf{e}} \ldots$ quantifiers.

The encoding function on types takes the form $[\![\tau]\!]_\Delta$ where Δ maps each type variable to its associated level variable, and it returns a pair $\tau'; \ell$ where ℓ is the universe level of τ':

$$
\begin{array}{rcll}
[\![t]\!]_\Delta & = & t\ ;\ \Delta(t) & \\
[\![\tau_1 \to \tau_2]\!]_\Delta & = & \tau_1' \xrightarrow{\mathsf{n}} \tau_2'\ ;\ \ell_1 \sqcup \ell_2 & \text{where } \tau_1'; \ell_1 = [\![\tau_1]\!]_\Delta \text{ and } \tau_2'; \ell_2 = [\![\tau_2]\!]_\Delta \\
[\![(t:*) \to \tau]\!]_\Delta & = & (l:\mathsf{Level}) \xrightarrow{\mathsf{e}} (t:\mathsf{Type}_l) \xrightarrow{\mathsf{n}} \tau'\ ;\ \ell' & \text{where } \tau'; \ell = [\![\tau]\!]_{\Delta,t:l} \text{ and } \ell' = 1 \sqcup \ell[0/l]
\end{array}
$$

Similarly the encoding function for terms takes the form $[\![e]\!]_\Delta$:

$$\begin{array}{rll}
[\![x]\!]_\Delta &= x & \\
[\![\lambda x{:}\tau \to e]\!]_\Delta &= \lambda t{:}\tau' \xrightarrow{\mathsf{n}} [\![e]\!]_\Delta & \text{where } \tau';\ell = [\![\tau]\!]_\Delta \\
[\![e_1\ e_2]\!]_\Delta &= [\![e_1]\!]_\Delta\ @^{\mathsf{n}} [\![e_2]\!]_\Delta & \\
[\![\lambda t{:}* \to e]\!]_\Delta &= \lambda l{:}\mathsf{Level} \xrightarrow{\mathsf{e}} \lambda t{:}\mathsf{Type}_l \xrightarrow{\mathsf{n}} [\![e]\!]_{\Delta,t:l} & \\
[\![e\ \tau]\!]_\Delta &= ([\![e]\!]_\Delta\ @^{\mathsf{e}} \ell) @^{\mathsf{n}} \tau' & \text{where } \tau';\ell = [\![\tau]\!]_\Delta
\end{array}$$

Finally we need to encode contexts as well, for which the encoding function takes the form $[\![\Gamma]\!]$ and it returns a pair $\Gamma';\Delta$:

$$\begin{array}{rll}
[\![\bullet]\!] &= \bullet\ ;\ \bullet & \\
[\![\Gamma, x{:}\tau]\!] &= \Gamma', x{:}[\![\tau]\!]_\Delta\ ;\ \Delta & \text{where } \Gamma';\Delta = [\![\Gamma]\!] \\
[\![\Gamma, t{:}*]\!] &= \Gamma', l{:}\mathsf{Level}, t{:}\mathsf{Type}_l\ ;\ \Delta, t{:}l & \text{where } \Gamma';\Delta = [\![\Gamma]\!]
\end{array}$$

▶ **Theorem 7** (EpCCω can encode System-F)**.**
For any $\Gamma \vdash e : \tau$ in System-F, we have $\Gamma' \vdash e' : \tau'$ and $\Gamma' \vdash \tau' : \mathsf{Type}_\ell$ in EpCCω where $\Gamma';\Delta = [\![\Gamma]\!]$, $e' = [\![e]\!]_\Delta$, and $\tau';\ell = [\![\tau]\!]_\Delta$.

Proof. By structural induction on the type derivation. ◀

4.5 The power of EpCCω

EpCCω seems to be flexible enough to cover most uses of impredicativity found in the context of programming, such as Church's encoding, Chlipala's parametric higher-order abstract syntax [10], typed closure representations, or iCAP [25]. It does so without restricting impredicativity to a single universe, and even makes those uses more flexible in EpCCω such as adding the equivalent of strong elimination in Church's encoding. So in this sense EpCCω is more powerful than systems like CCω.

Yet we have not even been able to generalize the above System-F encoding in order to encode arbitrary F_ω terms into EpCCω. For example, consider the following F_ω term:

$$\lambda t_1{:}* \to \lambda(t_2{:}* \to *) \to \lambda(x{:}t_2\ t_1) \to x$$

A simple encoding into EpCCω could be:

$$\lambda l{:}\mathsf{Level} \xrightarrow{\mathsf{e}} \lambda t_1{:}\mathsf{Type}_l \xrightarrow{\mathsf{n}} \lambda(t_2{:}\mathsf{Type}_l \xrightarrow{\mathsf{n}} \mathsf{Type}_l) \xrightarrow{\mathsf{n}} \lambda x{:}t_2 @^{\mathsf{n}} t_1 \xrightarrow{\mathsf{n}} x$$

But it's not faithful to the original F_ω term because it only preserves the impredicativity of the first λ. In order to get an encoding that can work for any F_ω term, we hence need an encoding which looks like:

$$\lambda l_1{:}\mathsf{Level} \xrightarrow{\mathsf{e}} \lambda t_1{:}\mathsf{Type}_{l_1} \xrightarrow{\mathsf{n}} \lambda l_2{:}\mathsf{Level} \xrightarrow{\mathsf{e}} \lambda t_2{:}T_2 \xrightarrow{\mathsf{n}} \lambda x{:}T_x \xrightarrow{\mathsf{n}} x$$

where T_2 refers to l_2. We can then choose T_2 and T_x as follows:

$$\begin{array}{l}
T_2 = (l_3{:}\mathsf{Level}) \xrightarrow{\mathsf{e}} \mathsf{Type}_{l_3} \xrightarrow{\mathsf{n}} \mathsf{Type}_{l_2} \\
T_x = t_2 @^{\mathsf{e}} l_1 @^{\mathsf{n}} t_1
\end{array}$$

This makes the term valid, but its semantics doesn't match that of the original F_ω term since we cannot pass the identity function $\lambda t{:}* \to t$ as f any more: its encoding would now have type $(l_3{:}\mathsf{Level}) \xrightarrow{\mathsf{e}} \mathsf{Type}_{l_3} \xrightarrow{\mathsf{n}} \mathsf{Type}_{l_3}$ instead of the expected $(l_3{:}\mathsf{Level}) \xrightarrow{\mathsf{e}} \mathsf{Type}_{l_3} \xrightarrow{\mathsf{n}} \mathsf{Type}_{l_2}$.

Similarly, we have not been able to adapt Hurkens's paradox to the EpCCω system either. Of course, all this says is that we do not know if EpCCω is consistent, but at least it indicates that this kind of impredicativity might be incomparable to the traditional form seen in CCω or λU^-.

5 Related work

In [3], Augustsson presents a language where inductive types only live in the bottom universe, and shows that everything from the higher universes can be erased. This is similar to our argument in Section 3.2, but with some important differences in the universe stratification and in the definition of erasure. His universe stratification is unusual in that it is designed to keep track of erasability and does not enforce predicativity, which makes it fundamentally very different. It turns out that for eCCω and eCIC, his stratification rules match our traditional rules when it comes to deciding if something is in the bottom universe, so his erasure should apply equally to a stratification like the one used here, although this is not the case when we consider systems like eCoq. More importantly, his notion of erasure is different from ours since his erasure of $(x:\tau_1) \xrightarrow{k} \tau_2$ is $\bullet$ meaning that it is significantly more permissive. For example, his erasure has to be *external* (i.e., performed after checking type convertibility), whereas the erasure we use here could be *internal*, as is the case in ICC [21] and ICC*[5].

In [30], Werner discusses *internal* erasure of Coq's impredicative Prop universe. This is done in the context of the proof-irrelevance kind of erasure, where Prop is restricted to be proof-irrelevant so that it can be erased from the non-Prop universes. So this approach is contrary to ours: we erase non-Prop arguments from Prop terms, whereas he erases Prop arguments from non-Prop terms. More importantly, this kind of erasure is already present in Coq, so what Werner proposes is to make it internal, that is to take advantage of this erasure to strengthen the convertibility rule during type checking, in the same way ICC [21] and ICC*[5] systems use a stronger convertibility rule to take advantage of the kind of erasure we use here, as discussed in Section 2.2. This strengthening comes at the cost of normalization, as shown by Abel and Coquand [1].

In [15], Gilbert et.al. present a Coq and Agda library which provides a similar internal erasure of proof-irrelevant propositions. In comparison to Werner's work, they use a slightly different definition of proof-irrelevance based on *mere propositions* [27] and they get internal erasure by construction rather than by adding it to they underlying system.

In [28], Uemura shows a model of a cubical λ-calculus with a bottom universe that is impredicative and admits univalence and shows it not to satisfy the propositional resizing axiom, which applies to proof-irrelevant propositions. This puts into question the consistency of this axiom in such a calculus.

6 Conclusion

We have taken a tour of the interactions between impredicativity and erasability of arguments in EPTS. We have shown that three of the five most well known systems that admit impredicativity do it in a way that implicitly constrains all impredicative abstractions and fields to be erasable (and that the remaining two almost do it as well). We have also shown that while impredicativity and erasability seem to be correlated, erasability is neither a necessary nor a sufficient condition for impredicativity to be consistent: the inconsistency of

$\lambda e U^-$ shows it's not sufficient, and our inability to show that UTT's impredicative definitions are all erasable suggests it's not necessary either.

It remains to be seen whether erasability as used in ECIC allows us to lift the restriction that strong elimination cannot be used on large inductive types without breaking consistency, and whether erasability as used in EpCCω allows us to introduce a form of impredicativity applicable to all universe levels without breaking consistency.

Another avenue of research might be to try and better understand the relationship between the kind of erasure of impredicatively quantified arguments discussed here and the impredicativity of proof-irrelevant terms, as used in Coq and in the propositional resizing axiom.

Acknowledgments

We would like to thank Chris League for his comments on earlier drafts of the paper, as well as the reviewers for their careful reading and very helpful feedback.

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 Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality, February 2020. Submitted to Logical Methods in Computer Science. URL: https://arxiv.org/abs/1911.08174.

2 Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1:29):1–36, 2012. doi:10.2168/LMCS-8(1:29)2012.

3 Lennart Augustsson. Cayenne—a language with dependent types. In *International Conference on Functional Programming*, page 239–250. ACM Press, September 1998. doi:10.1145/291251.289451.

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

5 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.

6 Bruno Bernardo. Towards an implicit calculus of inductive constructions. extending the implicit calculus of constructions with union and subset types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, August 2009. URL: https://hal.inria.fr/inria-00432649.

7 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22(2):1–46, 2012. doi:10.1017/S0956796812000056.

8 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.

9 Luca Cardelli. Phase distinctions in type theory. DEC-SRC manuscript, 1988. URL: https://pdfs.semanticscholar.org/4cb5/7987b78c5124bc0857155f99c11aa321546d.pdf.

10 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, Victoria, BC, September 2008. doi:10.1145/1411204.1411226.

11 Thierry Coquand. An analysis of Girard's paradox. In *Annual Symposium on Logic in Computer Science*, 1986. Also published as INRIA tech-report RR-0531. URL: https://hal.inria.fr/inria-00076023.

12 Thierry Coquand. A new paradox in type theory. In *Logic, Methodology, and Philosophy of Science*, pages 7–14, 1994. doi:10.1016/S0049-237X(06)80062-5.

13 Thomas Fruchart and Guiseppe Longo. Carnap's remarks on impredicative definitions and the genericity theorem. Technical Report LIENS-96-22, ENS, Paris, 1996. URL: ftp://ftp.di.ens.fr/pub/reports/liens-96-22.A4.ps.Z.

14 Herman Geuvers. (In)consistency of extensions of higher order logic and type theory. In *Types for Proofs and Programs*, pages 140–159, 2006. URL: https://www.cs.ru.nl/~herman/PUBS/inconsist-hol.pdf, doi:10.1007/978-3-540-74464-1_10.

15 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. In *Symposium on Principles of Programming Languages*, pages 3:1–3:28. ACM Press, 2019. doi:10.1145/3290316.

16 Eduardo Giménez. Codifying guarded definitions with recursive schemes. Technical Report RR1995-07, École Normale Supérieure de Lyon, 1994. URL: ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1995/RR1995-07.ps.Z.

17 J. Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, University of Paris VII, 1972. URL: https://pdfs.semanticscholar.org/e1a1/c345ce8ab4c11f176f1c42bcfc6a62ef4e3c.pdf.

18 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.

19 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.

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

21 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.

22 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.

23 Stefan Monnier. The Swiss coercion. In *Programming Languages meets Program Verification*, pages 33–40, Freiburg, Germany, September 2007. ACM Press. doi:10.1145/1292597.1292604.

662 **24** Stefan Monnier. Typer: ML boosted with type theory and Scheme. In *Journées Francophones*
663 *des Langages Applicatifs*, pages 193–208, 2019. URL: `https://hal.inria.fr/hal-01985195/`.

664 **25** Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *European*
665 *Symposium on Programming*, pages 149–168, 2014. URL: `https://cs.staff.au.dk/~birke/`
666 `papers/icap-conf.pdf`, `doi:10.1007/978-3-642-54833-8_9`.

667 **26** Matus Tejiscak. *Erasure in Dependently Typed Programming*. PhD thesis, University of St
668 Andrews, 2020.

669 **27** The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of*
670 *Mathematics*. Institute for Advanced Study, 2013. URL: `https://arxiv.org/abs/1308.0729`.

671 **28** Taichi Uemura. Cubical assemblies, a univalent and impredicative universe and a failure of
672 propositional resizing. In *Types for Proofs and Programs*, volume 130 of *Leibniz International*
673 *Proceedings in Informatics (LIPIcs)*, pages 7:1–7:20, 2019. `doi:10.4230/LIPIcs.TYPES.2018.`
674 `7`.

675 **29** Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université
676 Paris 7, Paris, France, 1994. URL: `https://hal.inria.fr/tel-00196524/`.

677 **30** Benjamin Werner. On the strength of proof-irrelevant type theories. *Logical Methods in*
678 *Computer Science*, 4(3):1–20, 2008. URL: `https://arxiv.org/abs/0808.3928`, `doi:10.1007/`
679 `11814771_49`.