On the role of type decorations in the Calculus of Inductive Constructions

Bruno Barras[1] and Benjamin Grégoire[2]

[1] INRIA Futurs, France Bruno.Barras@inria.fr
[2] INRIA Sophia-Antipolis, France Benjamin.Gregoire@sophia.inria.fr

Abstract. In proof systems like Coq [15], proof-checking involves comparing types modulo β-conversion, which is potentially a time-consuming task. Significant speed-ups are achieved by compiling proof terms, see [8]. Since compilation erases some type information, we have to show that convertibility is preserved by type erasure. This article shows the equivalence of the Calculus of Inductive Constructions (formalism of Coq) and its domain-free version where parameters of inductive types are also erased. It generalizes and strengthens significantly a similar result by Barthe and Sørensen [5] on the class of functional Domain-free Pure Type Systems.

1 Introduction

In proof systems based on the Curry-Howard isomorphism, proof-checking boils down to type-checking in a system with dependent types. Such systems usually include a conversion rule of the form:

$$\frac{\Gamma \vdash t : \tau \quad \tau \simeq_\beta \tau'}{\Gamma \vdash t : \tau'}[\text{conv}]$$

where $\simeq_\beta$ stands for β-convertibility. This rule can be used to make complex computation. Examples of that usage include reflection tactics in Coq and the proof of the four-colors theorem. This conversion rule is generally implemented in a purely interpretative way[1], because it is a hard task to perform strong β-reduction (reduction occurs also under binders) in a compiled setting. In [8] Grégoire and Leroy show how to strongly normalize and how to decide β-equivalence on terms, by compiling proof-terms towards an abstract machine (a slightly modified version of OCaml's ZAM) and analyzing computed values with a readback procedure.

This scheme raises a problem: compilation has the effect of erasing type annotations (used to ensure the decidability of type checking and so the impossibility of runtime error). So, while conversion is defined over Church-style terms (abstractions carry a type annotation $\lambda x{:}\tau.\, t$), the abstract machine based version of conversion works on "Curry-style" terms ($\lambda x.\, t$, also called *domain-free*

[1] By interpertative, we mean algorithms that perform the conversion test by explicitly manipulating proof terms represented as trees.

terms). The correctness of such compilation scheme with respect to the original formalism relies on a general issue of proving the equivalence of a given type system with its domain-free counterpart. This problem has already been studied in the case of Pure Type Systems [2] (PTS, Church-style) and their domain-free version, the Domain-Free Pure Type System [5] (DFPTS) by Barthe and Sørensen. The authors prove an equivalence theorem under the assumption that the system is normalizing (so we can reason on normal terms) and functional. The latter condition is used to ensure type uniqueness. Earlier, Streicher [14] proved this result for Calculus of Constructions, still based on normalization and type uniqueness.

Our paper enhances previous work in two ways. Firstly, we extend the results of [5] to a richer class of systems that feature cumulativity [2] and inductive types. A notable point of our notion of type erasure is that we erase parameters of constructors[3], since they do not participate in the computation. Our results apply to the Calculus of Inductive Constructions (CIC), and yield an efficient sound and complete convertibility test for Coq.

Secondly, our results do not rely on type uniqueness, which does not hold any more due to subtyping (even without subtyping, type uniqueness does not hold for any PTS). Instead, we introduce an equivalence on types to recover a loose notion of type uniqueness.

This equivalence theorem also has consequences on implementation. Many proof systems prefer to use Church-style λ-terms, in particular because type inference is decidable under simple conditions and it is often easier to build a set theoretic model of those formalisms. On the other hand, Curry-style terms reflect the computational behavior of λ-terms better. This is related to the fact that pure λ-calculus is the execution model of the core of many functional languages. But type inference is generally not decidable, and type checking fails on non-normal terms. The equivalence theorem shows that we can have a system with good properties such as type decidability, and compare terms as Curry-style terms, allowing compilation techniques. Regarding inductive types, parameters can be erased in constructors, which leads to the same representation as in a compiled language like OCaml [9].

We prove the equivalence between a type annotated system where conversion compares type decorations (we call it β) and a second type system (we call it ϵ) where annotations are in the syntax but conversion ignores them. Then it is trivial to define a third system (the domain free version) where type decorations are not in the syntax and then prove the equivalence with the system ϵ. This way, we separate the problems of changing the term representation and that of actually changing the conversion.

For explanatory purposes, we will distinguish the strengthening of Barthe's and Sørensen's result (removing the functionality hypothesis) and the extension

[2] cumulativity is a simple notion of subtyping that reduce need to duplicate definition across the various universes of the PTS.

[3] For instance, the first argument of the ternary constructor of lists $\texttt{cons} : \forall \alpha, \alpha \to \texttt{list}\ \alpha \to \texttt{list}\ \alpha$ can be erased.

to a broader class of systems. Section 2 introduces Cumulative Type Systems (CTS), which are PTS with cumulativity and an abstract notion of conversion. Then, we briefly give metatheoretical properties of CTS. Section 3 shows how both ω and ϵ systems can be represented by instantiating this abstract conversion in two ways. It ends by proving Preservation of Equational Theory (PET), Preservation of Subtyping (PS) and Preservation of Typing (PT) between both systems. These properties simply state that conversion, subtyping and typing are equivalent notions. Then, we will extend these results to inductive types (Sect. 4) and conclude.

2 A generic version of Cumulative Type Systems (CTS)

Pure Type Systems (PTS, [2]) are a generalization of several type systems such as simply typed λ-calculus, system F, Calculus of Constructions, etc. Since some systems have dependent types (type parameterized by expressions or programs), they use the same syntax for terms and types and types are also subject to a type discipline.

2.1 Syntax of terms

As for PTS, Cumulative Type Systems [3] are generated from specifications. To the three parameters of the PTS, we add two extra parameters. The first one $\prec$ allows subtyping over sorts: if $s_1 \prec s_2$, then any type of s_1 is also a type of s_2, without any explicit coercion. This is called cumulativity. The second extra parameter $\simeq$, called conversion, is a relation between types indicating which types are identified. In the rest of this paper we will instantiate this parameter with different relations. This follows the same idea as in [13, 11].

Let us make this more precise by simultaneously defining the syntax of terms ($\mathscr{T}$) and specifications of CTS ($\mathbf{S}$). Let $\mathscr{V}$ be an infinite set of variables.

Definition 1 (term and specification).
A specification is a tuple $\mathbf{S} = (\mathscr{S}, \mathscr{A}, \mathscr{R}, \prec, \simeq)$ *where*
- $\mathscr{S}$ *is a set of sorts.*
- $\mathscr{A} \subseteq \mathscr{S} \times \mathscr{S}$ *is a set of axioms.*
- $\mathscr{R} \subseteq \mathscr{S} \times \mathscr{S} \times \mathscr{S}$ *is a set of rules*
- $\prec \subseteq \mathscr{S} \times \mathscr{S}$ *is an inclusion relation between sorts.*
- $\simeq \subseteq \mathscr{T} \times \mathscr{T}$ *is an equivalence relation between terms. It should be a congruence:* $\forall x, M, N, N'.\ \ N \simeq N' \ \Rightarrow\ M\{x \leftarrow N\} \simeq M\{x \leftarrow N'\}$

The set $\mathscr{T}$ *of expressions (over* $\mathbf{S}$*) is given by the abstract syntax :*
$$\mathscr{T} ::= \mathscr{V} \mid \mathscr{S} \mid \Pi\mathscr{V}:\mathscr{T}.\,\mathscr{T} \mid \lambda\mathscr{V}:\mathscr{T}.\,\mathscr{T} \mid \mathscr{T}\ \mathscr{T}$$

We use t, A, B, M, N, T, U, V, etc. to denote elements of $\mathscr{T}$; x, y, z, etc. to denote elements of $\mathscr{V}$; s, s', etc. to denote elements of $\mathscr{S}$. The substitution of variable x for a term N in M will be written $M\{x \leftarrow N\}$. As usual, we consider β-reduction on terms, written $\rightarrow$. We write $\xrightarrow{*}$ its reflexive and transitive closure, and $\simeq_\beta$ the smallest equivalence relation including $\rightarrow$ (β-conversion). PTS are a special case of CTS where $\prec$ is $\emptyset$ and $\simeq$ is β-conversion.

2.2 Cumulativity

As already stated, cumulativity introduces some kind of subtyping. Let us now define the subtyping relation induced by our CTS parameters:

Definition 2 (cumulativity). *The one step subtyping relation $\preceq$ over a equivalence relation $\simeq$ and a inclusion relation between sort $\prec$ is given by the rules below.*

$$\frac{T \simeq U}{T \preceq U} \qquad \frac{s_1 \prec^* s_2}{s_1 \preceq s_2} \qquad \frac{T \simeq T' \qquad U \preceq U'}{\Pi x\!:\!T.U \preceq \Pi x\!:\!T'.U'}$$

This relation is also named cumulativity. We write $\preceq_S$ to refer to the equivalence relation and the inclusion relation between sort of S. When $\prec$ is fixed, we use as notation $\preceq_{\simeq}$ or just $\preceq$ if $\simeq$ is clear from the context.

Note that following Luo's Extended Calculus of Constructions [10], the subtyping relation is not contravariant w.r.t. the domain of functions (the domains of a function type and its subtype are convertible). Contravariance is rejected because it would invalidate our proof as we shall in section 3.3.

At that point, we define several properties of relations related to abstract rewriting systems.

Definition 3 (commutation, reducibility). *Let R_1, R_2 be two binary relations.*

- R_1, R_2 commute*, written $(R_1, R_2) \in \mathcal{C}$, iff*

$$\forall x, x_1, x_2.\ \ x\ R_1\ x_1\ \ \wedge\ \ x\ R_2\ x_2\ \ \Rightarrow\ \ \exists y.\ \ x_2\ R_1\ y\ \ \wedge\ \ x_1\ R_2\ y$$

- *R_1 is reducible to R_2 modulo β-reduction, written $R_1 \in \mathcal{R}_{R_2}$, iff*

$$\forall t, u.\ t\ R_1\ u \Rightarrow \exists t', u'.\ t \xrightarrow{*} t' \wedge u \xrightarrow{*} u' \wedge t'\ R_2\ u'$$

Lemma 1. *For any equivalence relation R, cumulativity preserves commutation with β-reduction*

$$(R, \xrightarrow{*}) \in \mathcal{C} \Rightarrow (\preceq_R, \xrightarrow{*}) \wedge (\preceq_R^{-1}, \xrightarrow{*}) \in \mathcal{C}$$

Lemma 2. *Cumulativity preserves reducibility to any equivalence relation commuting with β-reduction:*

$$(R_2, \xrightarrow{*}) \in \mathcal{C}\ \ \wedge\ \ R_1 \in \mathcal{R}_{R_2}\ \ \Rightarrow\ \ \preceq^*_{R_1} \in \mathcal{R}_{\preceq^*_{R_2}}$$

Proof : *See appendix A.*

2.3 Typing

Definition 4 (Typing judgment). *Let* **S** *be the specification $(\mathscr{S}, \mathscr{A}, \mathscr{R}, \prec, \simeq)$*

- *A context is a list:* $\Gamma\ ::=\ [\,]\ |\ \Gamma; (x\!:\!T)$
  $(x\!:\!T)$ *denotes a local declaration of a variable x of type T.*

$$\frac{}{\mathcal{WF}([])}[\mathrm{WE}] \qquad \frac{\Gamma \vdash T : s \qquad s \in \mathscr{S}}{\mathcal{WF}(\Gamma; (x\!:\!T))}[\mathrm{WS-LOCAL}]$$

$$\frac{\mathcal{WF}(\Gamma) \qquad (s_1,\ s_2) \in \mathscr{A}}{\Gamma \vdash s_1 : s_2}[\mathrm{SORT}] \qquad \frac{\mathcal{WF}(\Gamma) \qquad (x\!:\!T) \in \Gamma}{\Gamma \vdash x : T}[\mathrm{VAR}]$$

$$\frac{\Gamma \vdash T : s1 \qquad \Gamma; (x\!:\!T) \vdash U : s2 \qquad (s1,\ s2,\ s3) \in \mathscr{R}}{\Gamma \vdash \Pi x\!:\!T.U : s3}[\mathrm{PROD}]$$

$$\frac{\Gamma \vdash \Pi x\!:\!T.U : s \quad \Gamma; (x\!:\!T) \vdash M : U}{\Gamma \vdash \lambda x\!:\!T.M : \Pi x\!:\!T.U}[\mathrm{LAM}] \quad \frac{\Gamma \vdash M : \Pi x\!:\!T.U \quad \Gamma \vdash N : T}{\Gamma \vdash M\ N : U\{x \leftarrow N\}}[\mathrm{APP}]$$

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash U : s \qquad T \preceq^* U}{\Gamma \vdash M : U}[\mathrm{CONV}] \qquad \frac{\Gamma \vdash M : T \qquad T \preceq^* s}{\Gamma \vdash M : s}[\mathrm{CONV}_s]$$

Fig. 1. Typing rules for CTS

– *The typing relation $\vdash$ is given by the rules in Fig. 1. There is also a judgment $\mathcal{WF}()$ to mean that a context is well formed. Both two judgments are simultaneously defined by mutual induction. We occasionally write $\vdash_S$ to explicit the dependency with the specification* $\mathbf{S}$.

The rules are the same as for PTS except that in our CTS, CONV should rather be seen as a subsumption rule, and CONV_s is necessary when cumulativity is used towards a non-typable sort.

Figure 2 lists the fundamental meta-theoretical properties of CTS. They are easy generalizations of PTS's properties. First equation expresses that type derivations of CTS are preserved by substitution. The second one shows that typing is preserved by well-typed context narrowing. Equation (3), that a type is a sort or typable by a sort. Then we have the well-known subject reduction property. The last one is the inversion lemma. We will not give their proofs since they have been formally checked using Coq in [3][4].

Later on, we will study the relation between different CTS which differ on the inclusion between sorts and conversion. We say that $\mathbf{S}_1$ is included in $\mathbf{S}_2$ if they are included component-wise. In that case $\preceq_{\mathbf{S}_1} \subseteq \preceq_{\mathbf{S}_2}$ and $\vdash_{\mathbf{S}_1} \subseteq \vdash_{\mathbf{S}_2}$. Put it in another way, subtyping and typing are monotonic w.r.t. the specification.

3 β-conversion and conversion modulo type annotations

Now we have this general framework of CTS, we can instantiate it with the parameters corresponding to the considered logical formalisms. For the rest of this paper we suppose that $\mathscr{S}, \mathscr{A}, \mathscr{R}$ and $\prec$ are fixed. In the case of typeful systems, terms are identified modulo β:

[4] The complete source of that formalization is available online at `http://logical.inria.fr/~barras/pts_proofs/PTS/main.html`. All subsequent URLs will be relative to `http://logical.inria.fr/~barras/pts_proofs/PTS/`.

Substitution $\quad \Gamma \vdash N : T \wedge \Gamma;(x:T) \vdash M : U \Rightarrow \Gamma \vdash M\{x \leftarrow N\} : U\{x \leftarrow N\} \quad (1)$

`Metatheory.html#substitution`

Context conversion $\quad \Gamma \vdash M : T \ \wedge \ \Delta \preceq^* \Gamma \Rightarrow \Delta \vdash M : T \quad (2)$

`Metatheory.html#subtype_in_env`

Correctness of types $\quad \Gamma \vdash A : B \Rightarrow B \in \mathscr{S} \ \vee \ \exists s \in \mathscr{S}.\ \Gamma \vdash B : s \ (3)$

`Metatheory.html#type_correctness`

Subject reduction $\quad \Gamma \vdash t : T \ \wedge \ t \xrightarrow{*} t' \Rightarrow \Gamma \vdash t' : T \quad (4)$

`LambdaSound.html#beta_sound`

Inversion lemmas

$$
\begin{aligned}
\Gamma \vdash s_1 : T &\Rightarrow \exists s_2.\ (s_1,\ s_2) \in \mathscr{A} \wedge s_2 \preceq^* T \\
\Gamma \vdash x : T &\Rightarrow \exists T'.\ (x:T') \in \Gamma \wedge T' \preceq^* T \\
\Gamma \vdash \lambda x{:}A.\,M : T &\Rightarrow \exists B, s.\ \Gamma;(x:A) \vdash M : B \wedge \Gamma \vdash \Pi x{:}A.\,B : s \wedge \Pi x{:}A.\,B \preceq^* T \\
\Gamma \vdash M\ N : T &\Rightarrow \exists A, B.\ \Gamma \vdash M : \Pi x{:}A.\,B \wedge \Gamma \vdash N : A \wedge B\{x \leftarrow N\} \preceq^* T \\
\Gamma \vdash \Pi x{:}A.\,B : T &\Rightarrow \exists (s_1,\ s_2,\ s_3) \in \mathscr{R}.\ \Gamma \vdash A : s_1 \wedge \Gamma;(x:A) \vdash B : s_2 \wedge s_3 \preceq^* T
\end{aligned}
$$

`Metatheory.html#inversion_lemma`

Fig. 2. Meta-theoretical properties of CTS

Definition 5 (specification β). *Since β-conversion is a congruence, we can build a CTS upon it. Let β be the specification $(\mathscr{S}, \mathscr{A}, \mathscr{R}, \prec, \simeq_\beta)$.*

Lemma 3. *$\preceq^*_\beta$ is reducible to $\preceq_=$.*
Proof : *Since $\preceq_=$ is transitive, we only have to show $\preceq^*_\beta \ \in \mathcal{R}_{\preceq^*_=}$, which is a consequence of Lemma 2 and the well known Church-Rosser property of β-conversion.*

3.1 ϵ-conversion

To define the notion of convertibility that do not take type annotations into account we first define equality modulo type annotation. It captures the essence of domain-free conversion but within a type-carrying syntax.

Definition 6 (ϵ-equality, ϵ-convertibility). *Two terms are ϵ-equal if they are equal modulo type annotations. We write $=_\epsilon$ this equality. ϵ-convertibility is the smallest equivalence relation including ϵ-equality and β-reduction. We write $\simeq_\epsilon$ this relation.*

$$
\frac{}{x =_\epsilon x} \qquad \frac{}{s =_\epsilon s} \qquad \frac{T =_\epsilon T' \qquad U =_\epsilon U'}{T\ U =_\epsilon T'\ U'} \qquad \frac{T =_\epsilon T'}{\lambda x{:}A.\,T =_\epsilon \lambda x{:}A'.\,T'}
$$

$$
\frac{T =_\epsilon T' \qquad U =_\epsilon U'}{\Pi x{:}T.\,U =_\epsilon \Pi x{:}T'.\,U'} \quad \frac{T =_\epsilon U}{T \simeq_\epsilon U} \quad \frac{T \simeq_\epsilon U \qquad U \to V}{T \simeq_\epsilon V} \quad \frac{T \simeq_\epsilon U \qquad V \to U}{T \simeq_\epsilon V}
$$

Definition 7. *It is trivial to see that ϵ-equality and $\simeq_\epsilon$ are equivalence relations and congruences. Let ϵ be the specification $(\mathscr{S}, \mathscr{A}, \mathscr{R}, \prec, \simeq_\epsilon)$.*

This conversion enjoys reducibility results similar to β:

Lemma 4. $\simeq_\epsilon$ *is reducible to* $=_\epsilon$ *and* $\preceq_\epsilon^*$ *is reducible to* $\preceq_{=_\epsilon}$.
Proof : *First prove* $(=_\epsilon, \xrightarrow{*}) \in \mathcal{C}$ *and using the confluence of* β*-reduction, we extend the result to* $\simeq_\epsilon$. *For the second statement, we prove* $\preceq_\epsilon^* \in \mathcal{R}_{\preceq_{=_\epsilon}^*}$ *using Lemma 2, and remark that* $\preceq_{=_\epsilon}$ *is reflexive and transitive.*

3.2 Uniqueness of types

It is well known that any functional PTS (a PTS where $\mathscr{A}$ and $\mathscr{R}$ are functional relations) enjoys the type uniqueness property:

$$\Gamma \vdash M : T \;\wedge\; \Gamma \vdash M : T' \;\Rightarrow\; T \simeq_\beta T'$$

This has been already formally proved in Lego by Pollack [13]. Unfortunately, non trivial subtyping (including cumulativity) breaks this property. Let alone CTS, the property does not hold for non functional PTS, which include the well-known (and useful) Calculus of Constructions with universes.

However, we can remark that we only need type uniqueness regarding the *domain types* of functions. This relaxed uniqueness notion holds for CTS because subtyping can occur only on sorts in the codomain (as Luo already noticed for the Extended Calculus of Constructions [10]). This uniqueness of domain types is formalized by a relation $\approx_\beta$ which ensures convertibility of domain types, but allows any change of sort in the codomain. It reuses Definition 2.

Definition 8. *We write* $\approx_\beta$ *the reflexive and transitive closure of the cumulativity relation derived from* $\simeq_\beta$ *and* $\mathscr{S} \times \mathscr{S}$, *and* $\approx_=$ *the cumulativity relation derived from* $\mathscr{S} \times \mathscr{S}$ *and* $=$. *We say that* t_1 *is close to* t_2 *if* $t_1 \approx_\beta t_2$.

The important facts are $\approx_\beta$ is an equivalence relation, $\approx_=$ is transitive and $\approx_\beta$ is reducible to $\approx_=$ (Lemma 2).

Lemma 5 (type uniqueness modulo $\approx_\beta$). *Specification* β *has type uniqueness modulo* $\approx_\beta$*:*

$$\Gamma \vdash_\beta t : T \;\wedge\; \Gamma \vdash_\beta t : T' \;\Rightarrow\; T \approx_\beta T'$$

Proof : *by induction on* $\Gamma \vdash_\beta t : T$, *then inversion on* $\Gamma \vdash_\beta t : T'$. *In all cases, we use the fact that* $\preceq_\beta^* \subset \approx_\beta$ *(by monotonicity) and that* $\approx_\beta$ *is a symmetric and transitive relation. Application case uses the fact that* $\Pi x{:}T.U \approx_\beta \Pi x{:}T'.U' \Rightarrow T \simeq_\beta T' \;\wedge\; U \approx_\beta U$.

3.3 Equivalence of ϵ and β

Proving $\Gamma \vdash_\beta t : T \Rightarrow \Gamma \vdash_\epsilon t : T$ is trivial by monotonicity of typing. The converse is more difficult to derive. The first idea is to proceed by induction on the typing judgment. The only difficulty is with the conversion rules. It is of course false that $\preceq_\epsilon^*$ is included in $\preceq_\beta^*$, even for well typed terms: take $\lambda x{:}A.\,x : A \to A$ and $\lambda x{:}B.\,x : B \to B$. So we have to do some induction loading. We can remark that if we compare only objects of same type, then we would necessarily have $A \simeq_\beta B$ (note that it would not be the case if cumulativity was contravariant w.r.t. the domain of functions). In order to have the weakest invariant we only assume that their types are close. But this invariant has to propagate to subterms. Consider $c : C$ and

$$(\lambda f\!:\!A \to A.\, c)\ (\lambda x\!:\!A.\, x) \qquad (\lambda f\!:\!B \to B.\, c)\ (\lambda x\!:\!B.\, x).$$

Both terms have type C, but not their subterms: arguments respectively have type $A \to A$ and $B \to B$. This example shows that β-redexes break the proposed invariant. If we assume our terms are in normal form, λ-abstractions are found only as argument of a variable. Since two ϵ-equal variables are equal, domain type uniqueness can establish the invariant that ϵ-convertible abstraction are compared only when we know their types are convertible, hence close. So, we can prove:

Lemma 6.

$$\left.\begin{array}{ll} \Gamma \vdash_\beta t : T & \Gamma \vdash_\beta t' : T' \\ t =_\epsilon t' & t, t' \in \mathcal{NF} \\ t \notin \lambda \ \vee\ T \approx_\beta T' & \end{array}\right\} \Rightarrow t = t'$$

where λ is the set of lambda abstraction terms.
Proof : *by induction on t, inversion of hypotheses $t =_\epsilon t'$, $\Gamma \vdash_\beta t : T$ and $\Gamma \vdash_\beta t' : T'$. We do only the interesting cases.*

- *Cases for sorts, variables and products are trivial.*
- *Case $t = \lambda x\!:\!A.\, M$ we have $t' = \lambda x\!:\!A'.\, M'$; $A, M, A', M' \in \mathcal{NF}$; $M =_\epsilon M'$. Inversion of type judgments yields:*
  $\Gamma \vdash_\beta \Pi x\!:\!A.\, B : s$; $\quad \Gamma; (x\!:\!A) \vdash_\beta M : B$;
  $\Gamma \vdash_\beta \Pi x\!:\!A'.\, B' : s'$; $\quad \Gamma; (x\!:\!A') \vdash_\beta M' : B'$;
  $\Pi x\!:\!A.\, B \preceq_\beta^* T \quad \Pi x\!:\!A'.\, B' \preceq_\beta^* T'$
  Thanks to last premise, we get $\Pi x\!:\!A.\, B \approx_\beta \Pi x\!:\!A'.\, B'$, so $A \simeq_\beta A'$ and $B \approx_\beta B'$. Since A, A' are in normal form, they are equal (Lemma 3). Equality of bodies holds using the induction hypothesis.
- *Case $t = M\ N$ and $t' = M'\ N'$; inversion of type judgments yields: $\Gamma \vdash_\beta M : \Pi x\!:\!A.\, B$; $\Gamma \vdash_\beta N : A$; $\Gamma \vdash_\beta M' : \Pi x\!:\!A'.\, B'$; $\Gamma \vdash_\beta N' : A'$*
  Since $t \in \mathcal{NF}$, M is not a abstraction, so by induction hypothesis $M = M'$. By uniqueness of type (Lemma 5) $\Pi x\!:\!A.\, B \approx_\beta \Pi x\!:\!A'.\, B'$ so $A \simeq_\beta A'$. Equality of arguments holds using the induction hypothesis.

The invariant is weaker than in Barthe and Sørensen [5], and leads to a simpler proof.

Theorem 1 (PET ϵ w.r.t. β). *If specification β is normalizing, $\Gamma \vdash_\beta M : T$ and $\Gamma \vdash_\beta M' : T$,*

$$M \simeq_\beta M' \Leftrightarrow M \simeq_\epsilon M'$$

Proof : *$M \simeq_\beta M' \Rightarrow M \simeq_\epsilon M'$ holds by monotonicity. Now assume $M \simeq_\epsilon M'$. Since specification β is normalizing, M (resp. M') has a normal form N (resp. N') which has type T by subject reduction. We have $N \simeq_\epsilon N'$ and also $N =_\epsilon N'$ by reducibility. (Lemma 4). Lemma 6 proves $N = N'$, so $M \simeq_\beta M'$.*

In fact, we can replace the two premises of this theorem by $\Gamma \vdash_\beta M : T$ and $\Gamma \vdash_\beta M' : T'$ and $T \approx_\beta T'$. We only need $T \approx_\beta T'$ to apply Lemma 6.

We extend Lemma 6 to the cumulative subtyping relation:

Lemma 7.

$$\left.\begin{array}{ll} \Gamma \vdash_\beta t : T & \Gamma \vdash_\beta t' : T' \\ t \preceq_{=_\epsilon} t' & t, t' \in \mathcal{NF} \\ t \notin \lambda \;\vee\; T \approx_\beta T' & \end{array}\right\} \Rightarrow t \preceq_{=} t'$$

Proof : *By induction over $t \preceq_{=_\epsilon} t'$. Obviously we use Lemma 6.*

Corollary 1 (PS ϵ w.r.t. β). *If specification β is normalizing, $\Gamma \vdash_\beta T : s$ and $\Gamma \vdash_\beta U : s'$, then*

$$T \preceq^*_\epsilon U \Leftrightarrow T \preceq^*_\beta U$$

Note that the normalization hypothesis is required for the same reason as for Theorem 1.

Lemma 8. *If specification β is normalizing then*

$$\Gamma \vdash_\epsilon t : T \;\Rightarrow\; \Gamma \vdash_\beta t : T \text{ and } \mathcal{WF}_\epsilon(\Gamma) \;\Rightarrow\; \mathcal{WF}_\beta(\Gamma)$$

Proof : *by mutual induction over $\Gamma \vdash_\epsilon t : T$, all cases are trivial but (CONV) and (CONV$_s$).*

- *Case of (CONV$_s$): by induction hypothesis $\Gamma \vdash_\beta t : T$. Reducibility property (Lemma 4) yields $T \xrightarrow{*} T' \preceq_{=_\epsilon} s$, so by inversion $T \xrightarrow{*} T' = s' \prec s$, we conclude $T \preceq^*_\beta s$*
- *Case of (CONV): by induction hypothesis we get $\Gamma \vdash_\beta t : T$ and $\Gamma \vdash_\beta U : s$. By correctness of type, either T is a sort s_1 and by a argument similar to (CONV$_s$) we prove $s_1 \prec s' \xleftarrow{*} U$ and conclude, or there exists a sort s' such that $\Gamma \vdash_\beta T : s'$. Preservation of subtyping entails $T \preceq^*_\beta U$ and we can conclude.*

Theorem 2 (PT ϵ w.r.t. β). *If specification β is normalizing then*

$$\Gamma \vdash_\epsilon t : T \Leftrightarrow \Gamma \vdash_\beta t : T$$

4 Extension to Calculus of Inductive Constructions

The goal of this section is to extend preservation of typing results about CTS to the Calculus of Inductive Constructions (CIC). The extra features are inductive types, which are a generalisation of ML's datatypes. To be precise, CIC is not parameterized by a sort hierarchy, but since the latter has very few impact on the syntactic metatheory, we do not define it, but rather use it abstractly. See [15] for a precise definition.

The proof follows exactly the same steps, so we will only mention places where there are additional cases. Since we still consider conversion as a parameter we will be able to share many properties between the usual CIC and its ϵ counterpart. Let us first define the syntax, reduction rules and typing rules of this common core.

4.1 Syntax of CIC

Inductive types allow to build (well founded) data structures with variants using *constructors*. It is also possible to analyze variants and access constructors arguments by shallow *pattern-matching*. Finally, there is a facility to build recursive functions (*fixpoints*). Some care is needed not to break the logical consistency of the formalism.

Definition 9 (Terms and specifications of CIC). *Let $\mathscr{I}$ be a set of names. We extend expressions with inductive constructions:*

$$\textit{Terms} : \mathscr{T} ::= \ldots \mid \mathscr{I} \mid \mathsf{C}^i_{\mathscr{I}}(\overrightarrow{\mathscr{T}}, \overrightarrow{\mathscr{T}}) \mid \langle \mathscr{T} \rangle \mathsf{case}\ \mathscr{T}\ \mathsf{of}\ \overrightarrow{\overrightarrow{\mathscr{V}} \Rightarrow \mathscr{T}} \mid \mathsf{fix}_n(\mathscr{V} : \mathscr{T} := \mathscr{T})$$

We use I to denote elements of $\mathscr{I}$. Notation $\overrightarrow{X}$ denotes a vector of Xs ($\#(\overrightarrow{v})$ is the length of $\overrightarrow{v}$).

Specifications have a sixth field ELIM $\subseteq \mathscr{I} \times \mathscr{S}$ *that controls the range of pattern-matching for each inductive type.*

Set $\mathscr{I}$ is the set of names of inductive types (e.g. `list`, `prod`, etc.). Constructors are not identified by name, but by a couple formed of the inductive type it belongs to, and a number identifying which variant it builds: $\mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a})$ is the i-th constructor of inductive type I. Since they represent datastructures, we enforce that they are always fully applied to arguments ($\overrightarrow{a}$). Vector $\overrightarrow{p}$ is the value of the parameters, they can be thought of as the explicit instantiation of polymorphic parameters of ML datatypes. They are syntactically separated from "real arguments" for convenience. A built-in **case** construct allows shallow pattern-matching on terms of inductive types, in the construction $\langle P \rangle \mathsf{case}\ M\ \mathsf{of}\ \overrightarrow{\overrightarrow{x} \Rightarrow b}$, M is the term to destruct, $\overrightarrow{x_i}$ are bounded variables for each branch b_i, and denote the arguments of the i-th constructor. P is called a *predicate* and is here only to ensure decidability of type-checking in the case of dependent elimination. This will be explained later on.

The reduction rule for **case** construct allows to select the branch corresponding to the constructor of an object. If the latter is a constructor then a reduction can occur:

$$\langle P \rangle \mathsf{case}\ \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a})\ \mathsf{of}\ \overrightarrow{\overrightarrow{x} \Rightarrow t} \ \rightarrow\ t_i\{\overrightarrow{x}_i \leftarrow \overrightarrow{a}\} \qquad \text{if } \#(\overrightarrow{x}_i) = \#(\overrightarrow{a})$$

where $t\{\overrightarrow{x} \leftarrow \overrightarrow{a}\}$ is the parallel substitution of terms $\overrightarrow{a}$ for variables $\overrightarrow{x}$ in t.

Note that P, I and $\overrightarrow{p}$ do not participate in the reduction, so they would be erased at compile-time. We will show that I and $\overrightarrow{p}$ can be erased, but not P.

Finally, the calculus supports recursive functions via guarded fixpoints

$$\mathsf{fix}_n(f : T := M)$$

T represents the type of the fixpoint, M its body; f is the name of the variable used in M to make recursive calls, and n is the position of the recursive argument. The usual reduction rule for fixpoints is $F \rightarrow M\{f \leftarrow F\}$ for

$F = \mathsf{fix}_n(f : T := M)$, but such definition instantly breaks strong normalization. To avoid infinite unrolling of the fixpoint, reduction is allowed only when the n-th argument is in constructor form. This *guard* associated to a typing condition that ensures that M makes a structural recursion over its n-th argument will preserve normalization. The guarded reduction is

$$F\ \overrightarrow{t} \;\rightarrow\; (M\{f \leftarrow F\})\ \overrightarrow{t} \qquad \text{if } \#(\overrightarrow{t}) = n \wedge t_n = \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a}) \wedge F = \mathsf{fix}_n(f : T := M)$$

Here we can also remark that T does not participate in the reduction, but as for the case predicate, type of fixpoints will not be erasable.

4.2 Typing

Before defining the typing rules of the inductive constructions, we introduce a new judgment $\Gamma \vdash T \ @\ \overrightarrow{u} \ \triangleright\ A$ to type-check n-ary applications . It should read: in context Γ, an expression of type T can be applied to arguments $\overrightarrow{u}$, and this application has return type A.

In traditional presentation of CIC, typing rules are configured by a signature Σ which contains declarations of inductive types, that is a family name I with his type and a type for each constructor of I.

Definition 10 (signature).

$$\Sigma ::= [] \mid \Sigma; \mathit{Ind}(I[\Delta_p] : \Pi\Delta_a.\, s := \overrightarrow{\Pi\Delta_i.\, I\ \mathit{Dom}(\Delta_p)\ \overrightarrow{t_i}})$$

Context Δ_p declares the parameters of the inductive definition. They are global to the definition and constructor can refer to them. Context Δ_a is the type of "real" arguments of I. s is the sort where the inductive objects lie. Then for each constructor, Δ_i gives the type of arguments of the i-th constructor. The inductive name I may appear in Δ_i. Finally, $\overrightarrow{t_i}$ defines which instance of the "real" arguments of I the constructor builds.

The same way contexts are subject to a typing judgment $\mathcal{WF}()$, there is a judgment to check that inductive declarations are well-formed. It includes type-checking of the various components of the declaration and a syntactic criterion called *positivity* to ensure strong normalization and consistency, but its exact definition does not matter here. See [12] for details.

Definition 11 (typing of CIC). *Typing rules for CTS are extended with the new rule defined in Fig. 3.*

Rule (IND) is like that of variables. Rule (CONSTR) is a combination of a variable rule (i-th constructor has type $\Pi\Delta_p.\, T_i$) and n-ary application (we do as if it was applied to $\overrightarrow{p}\,\overrightarrow{a}$). The side condition ensures that parameters and arguments are splitted correctly. Rule (FIX) is as usual except there is a side condition (GUARDED) that ensures that the fixpoint proceeds by structural induction over its n-th argument. It is a syntactic criterion we will not detail here.

$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash A \mathbin{@} [\,] \,\triangleright\, A}[\text{VNIL}] \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash U\{x \leftarrow t\} \mathbin{@} \overrightarrow{u} \,\triangleright\, A}{\Gamma \vdash \Pi x{:}T.\, U \mathbin{@} t\overrightarrow{u} \,\triangleright\, A}[\text{VCONS}]$$

$$\frac{\begin{array}{c}\mathcal{WF}(\Gamma)\\ \mathsf{Ind}(I[\Delta_p] : A := \overrightarrow{T}) \in \Sigma\end{array}}{\Gamma \vdash I : \Pi\Delta_p.\, A}[\text{IND}] \qquad \frac{\begin{array}{c}\mathsf{Ind}(I[\Delta_p] : A := \overrightarrow{T}) \in \Sigma\\ \#(\Delta_p) = \#(\overrightarrow{p})\\ \Gamma \vdash \Pi\Delta_p.\, T_i \mathbin{@} \overrightarrow{p}\,\overrightarrow{a} \,\triangleright\, I\ \overrightarrow{p}\ \overrightarrow{u}\end{array}}{\Gamma \vdash \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a}) : I\ \overrightarrow{p}\ \overrightarrow{u}}[\text{CONSTR}]$$

$$\frac{\begin{array}{c}\mathsf{Ind}(I[\Delta_p] : \Pi\Delta_a.\, s := \overrightarrow{\Pi\Delta_i.\, I\ \mathrm{Dom}(\Delta_p)\ \overrightarrow{t_i}}) \in \Sigma \quad \Gamma \vdash M : I\ \overrightarrow{p}\ \overrightarrow{a}\\ \textsc{Elim}(I, s') \qquad \forall i.\ \overrightarrow{x_i} = \mathrm{Dom}(\Delta_i)\\ \Gamma \vdash P : \Pi\Delta_a\{\mathrm{Dom}(\Delta_p) \leftarrow \overrightarrow{p}\}.\, \Pi x{:}I\ \overrightarrow{p}\ \mathrm{Dom}(\Delta_a).\, s'\\ \forall i.\ \Gamma\Delta_i\{\mathrm{Dom}(\Delta_p) \leftarrow \overrightarrow{p}\} \vdash b_i : P\ \overrightarrow{t_i}\{\mathrm{Dom}(\Delta_p) \leftarrow \overrightarrow{p}\}\ \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{x_i})\end{array}}{\Gamma \vdash \langle P\rangle \mathsf{case}\ M\ \mathsf{of}\ \overrightarrow{\overrightarrow{x} \Rightarrow b} : P\ \overrightarrow{a}\ M}[\text{CASE}]$$

$$\frac{\Gamma; (f{:}T) \vdash M : T \qquad \textsc{Guarded}(\mathsf{fix}_n(f{:}T := M))}{\Gamma \vdash \mathsf{fix}_n(f{:}T := M) : T}[\text{FIX}]$$

Fig. 3. New Typing rules for CIC

Rule (CASE) is the most complicated. Because of dependent types, branches may have different types. The type of the i-th branch is equal to P instantiated with the particular instance of the i-th constructor. And the type of the expression is P instantiated with the instance of the matched term (M). Side condition $\textsc{Elim}(I, s')$ is used to restrict the class of objects that can be built by pattern-matching. It may be necessary to be restricitve to avoid paradoxes. However, its definition is not relevent to our purposes.

The metatheory of CIC has been studied by various authors. It was first introduced by Paulin [12]. Substitution lemma, type correctness, subject reduction and type uniqueness also hold. Inversion lemma has to be extended to the case of the inductive constructions. We do not define it here but it always follow the same scheme. In his Ph.D., Barras [3] formalized the syntactic metatheory (strong normalization excluded) of an alternative presentation of CIC in Coq[5]. In particular, CIC enjoys the type uniqueness property modulo $\approx_\beta$ (the proof is the same as 5).

Hypothesis 1 (strong normalization) *CIC is normalizing.*

The above hypothesis can be seen either as a claim that CIC is normalizing (Werner [16] showed the strong normalization of CIC but with a subset of the sort hierarchy[6]) or as an assumption on the sort hierarchy for the subsequent lemmas to hold, if we see CIC as a general framework parameterized like CTSs.

[5] Of course, Gödel's second incompleteness theorem shows that if CIC is consistent, it is not possible to show this consistency within CIC.

[6] Yet the trickiest part: it includes non degenerated impredicativity and strong elimination...

4.3 ϵ-equality

For CIC, apart from erasing domain types of functions, ϵ-equality also ignores parameters and inductive names of constructors; the only relevant information for constructors are their constructor number and *real arguments*. As before we can ensure the equality of the erased part of constructors from the equality of their types. For instance, `0` and `false` are convertible and can have the same representation (their constructor number 0). Moreover, lists `(cons nat 0 (nil nat))` and `(cons bool false (nil bool))` are also convertible because their parameters (here the polymorphic arguments `nat` and `bool`) are ignored. This is what we call *Calculus of Inductive Constructions with Implicit Parameters*.

In this calculus, the conversion algorithm can safely implement constructors by a pair formed with a constructor number and a list of *real arguments*. It worths mentioning that it corresponds pretty well to how datatypes are compiled in languages of the ML family.

Definition 12 (ϵ-equality). *We extend ϵ-equality and ϵ-convertibility (Def. 6) with the following rules:*

$$\frac{}{I =_\epsilon I} \qquad \frac{P =_\epsilon P' \qquad M =_\epsilon M' \qquad \forall i,\ \overrightarrow{x_i} = \overrightarrow{x'_i} \qquad t_i =_\epsilon t'_i}{\langle P\rangle \mathsf{case}\ M\ \mathsf{of}\ \overrightarrow{\overrightarrow{x} \Rightarrow t} =_\epsilon \langle P'\rangle \mathsf{case}\ M'\ \mathsf{of}\ \overrightarrow{\overrightarrow{x'} \Rightarrow t'}}$$

$$\frac{i = i' \qquad \overrightarrow{a} =_\epsilon \overrightarrow{a'}}{\mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a}) =_\epsilon \mathsf{C}^{i'}_{I'}(\overrightarrow{p'}, \overrightarrow{a'})} \qquad \frac{T =_\epsilon T' \qquad M =_\epsilon M'}{\mathsf{fix}_n(f : T := M) =_\epsilon \mathsf{fix}_n(f : T' := M')}$$

Remark that we do not erase type information of fixpoints and cases. This is because it breaks Preservation of Equational Theory. For example terms

$$\begin{array}{l}\mathsf{fix}_2(f : (B \to B) \to A \to A := \lambda g : B \to B.\ \lambda x : A.\ x)\ \lambda x : B.\ x \\ \mathsf{fix}_2(f : (C \to C) \to A \to A := \lambda g : C \to C.\ \lambda x : A.\ x)\ \lambda x : C.\ x\end{array}$$

have the same type, are in normal form (they have no second argument), and are not convertible, but would become equal if we ignore information of fixpoints. The key point is that guarded fixpoints can behave as a non-reducible β-redex (when partially applied or when recursive argument is not a constructor). We can find some similar counter-examples where a non-reducible case blocks a β-redex, so we cannot ignore case's predicate.

4.4 Equivalence of CICϵ w.r.t. CIC

In Sect. 3 the proof relies on the ability to first infer the type of a head term in normal form and second to verify the convertibility of abstractions with close types. As a preliminary, we can extend the result of uniqueness of typing. And as for product, we have a kind of inversion for close inductive types :

$$I\ \overrightarrow{a} \approx_\beta I'\ \overrightarrow{a'} \Rightarrow I = I' \wedge \overrightarrow{a} \simeq_\beta \overrightarrow{a'}$$

The premise regarding the type constraint is changed since we must know that the types must be $\approx_\beta$ also in the case of constructors. Firstly regarding ϵ-equal normal forms:

Lemma 9.
$$\left.\begin{array}{ll} \Gamma \vdash_\beta t : T & \Gamma \vdash_\beta t' : T' \\ t =_\epsilon t' & t, t' \in \mathcal{NF} \\ t \notin \{\lambda, C\} \ \vee \ T \approx_\beta T' & \end{array}\right\} \Rightarrow t = t'$$

where C is the set of constructor terms.
Proof : *The interesting cases are those for constructors and pattern-matching (see appendix B for a detailed proof). Convertibility of constructors do not imply convertibility of their parameters, but last premise entails that their types are close (as for abstractions), so they belong to the same inductive type with the same parameters.*

For pattern-matching, we first prove the equality of arguments by induction hypothesis, which can be neither a constructor (t is in normal form) nor an abstraction (t is well-typed). By uniqueness of types, their types are close, which implies that both t and t' are pattern-matching over the same inductive with same parameters. So, both predicates have close types. By induction hypothesis, they are equal. So branches have close types pairwise. Finally we can prove the equality of branches.

Again, the rest of the proof goes exactly as in Section 3.3, and we can conclude to the equivalence of both systems:

Theorem 3 (PET,PS,PT for CIC). *If specification β is normalizing then*

$$\begin{array}{ll} \textit{(PET)} & \Gamma \vdash_\beta M : T \wedge \Gamma \vdash_\beta M' : T \ \Rightarrow \ M \simeq_\beta M' \Leftrightarrow M \simeq_\epsilon M' \\ \textit{(PS)} & \Gamma \vdash_\beta T : s \wedge \Gamma \vdash_\beta U : s' \ \Rightarrow \ T \preceq^*_\epsilon U \Leftrightarrow T \preceq^*_\beta U \\ \textit{(PT)} & \Gamma \vdash_\epsilon t : T \Leftrightarrow \Gamma \vdash_\beta t : T \end{array}$$

It is easy to show that CIC_ϵ is equivalent to is "the Calculus of Inductive Constructions with Implicit Parameters" (defined has CIC where type decorations and inductive parameters are remove from the syntax).

5 Conclusion and Future Work

We have introduced an (almost) type-free version of the Calculus of Inductive Constructions. In this new formalism, conversion test is more efficient, and moreover is compatible with compilation of proof terms as in [8]. We have shown that it is equivalent to CIC (provided that the latter is normalizing), by generalizing Barthe's and Sørensen's proof [5]. We can not get rid the normalization hypothesis since, as shown Barthe and Coquand [4], system U^- (a non normalizing PTS) is not equivalent to its domain-free version.

Our equivalence proof can be turned into an algorithm that reannotates a type-free term in normal form given its (unannotated) type. This is useful for toplevels to display the result of a normalization step. This algorithm has been integrated to the current development version of Coq.

A first direction to investigate is what happens if we remove the predicate of pattern-matching expressions and the type of fixpoints. We have shown that preservation of typing does not hold in the way we stated it. Nonetheless, it would be interesting to see how the formalism is affected regarding for instance expressivity and consistency.

Another direction is to study the case of contravariant subtyping. The problem here is that contravariance breaks our type uniqueness property and so our main lemma 6. So, again, equational theory is not exactly preserved, but we conjecture the equiconsistency of both systems. However, contravariant subtyping may radically change the way the proof works, so let us mention that adding subtyping to depend types has already been studied by Aspinall and Compagnoni [1] and by Castagna and Chen [6], and by Chen [7] for the Calculus of Constructions. We might need some of the proof techniques developed there.

References

1. D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.
2. H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Abramsky & Gabbay & Maibaum (Eds.), Clarendon, 1992.
3. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.
4. G. Barthe and T. Coquand. On the equational theory of non-normalizing pure type systems. *Journal of Functional Programming*, 14(2):191–209, Mar. 2004.
5. G. Barthe and M. Sørensen. Domain-free pure type systems. *In Journal of Functional Programming*, 10(5):412–452, September 2000.
6. Castagna and Chen. Dependent types with subtyping and late-bound overloading. *INFCTRL: Information and Computation (formerly Information and Control)*, 168, 2001.
7. G. Chen. Subtyping calculus of construction. In *22nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 1997.
8. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.
9. X. Leroy and J. V. D. Doligez. *The Objective Caml System*. Institut National de Recherche en Informatique et en Automatique, August 2004. Software and documentation available on the Web, http://caml.inria.fr/.
10. Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
11. J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4), Nov. 1999.
12. C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Ph.d. thesis, Paris 7, January 1989.
13. R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994.
14. T. Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Birkhauser, 1991.
15. The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, march 2002. http://coq.inria.fr/doc8/main.html.
16. B. Werner. *Une Théorie de Constructions Inductives*. Ph.d. thesis, Université Paris 7, May 1994.

A Proof of Lemma 2

First we prove by induction on $\preceq_{R_1}$

$$R_1 \in \mathcal{R}_{R_2} \Rightarrow \preceq_{R_1} \in \mathcal{R}_{\preceq_{R_2}} \qquad (*)$$

Then by induction on the number of steps in $T \preceq^*_{R_1} U$. The base cases are trivial. The inductive case is explained in the following diagram:

$$\begin{tikzcd}
 & T \arrow[dl, "*"'] \arrow[r, phantom, "\preceq_{R_1}"] \arrow[d, phantom, "(2)"] & V \arrow[dl, "*"] \arrow[dr, "*"] \arrow[r, phantom, "\preceq^*_{R_1}"] & U \arrow[dr, "*"] \arrow[d, phantom, "(1)"] & \\
T' \arrow[dr, "*"'] \arrow[r, phantom, "\preceq_{R_2}"] & V'' \arrow[dr, "*"] \arrow[rr, phantom, "(3)"] \arrow[d, phantom, "(5)"] & & V' \arrow[dl, "*"'] \arrow[r, phantom, "\preceq^*_{R_2}"] \arrow[d, phantom, "(4)"] & U' \arrow[dl, "*"] \\
 & T'' \arrow[r, phantom, "\preceq_{R_2}"] & W \arrow[r, phantom, "\preceq^*_{R_2}"] & U'' &
\end{tikzcd}$$

(1) existence of V', U' by induction hypothesis
(2) existence of T', V'' by (*)
(3) existence of W by confluence of β-reduction
(4) existence of U'' by $(R_2, \xrightarrow{*}) \in \mathcal{C}$ and Lemma 1
(5) existence of T'' by $(R_2, \xrightarrow{*}) \in \mathcal{C}$, Lemma 1 and symmetry of R_2

B Proof of Lemma 9

By mutual induction on t and $\overrightarrow{u}$. The property for the n-ary application judgements is:

$$\left.\begin{array}{ll} \Gamma \vdash_\beta T \mathbin{@} \overrightarrow{u} \rhd A & \Gamma \vdash_\beta T \mathbin{@} \overrightarrow{u'} \rhd A \\ \overrightarrow{u} =_\epsilon \overrightarrow{u'} & \overrightarrow{u}, \overrightarrow{u'} \in \mathcal{NF} \end{array}\right\} \Rightarrow \overrightarrow{u} = \overrightarrow{u'}$$

and is easily proven from induction hypotheses. We treat only cases specific to CIC:

- Case of inductive is trivial, for fixpoints just uses induction hypotheses.
- Case of $t = \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a})$, we have $t' = \mathsf{C}^i_{I'}(\overrightarrow{p'}, \overrightarrow{a'})$, $\overrightarrow{a} =_\epsilon \overrightarrow{a'}$. By inversion on $\Gamma \vdash_\beta \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{a}) : T$ and $\Gamma \vdash_\beta \mathsf{C}^i_{I'}(\overrightarrow{p'}, \overrightarrow{a'}) : T'$ we have:
  $I\ \overrightarrow{p}\ \overrightarrow{u} \preceq^*_\beta T,\ \#(\Delta_p) = \#(\overrightarrow{p}),\ I'\ \overrightarrow{p'}\ \overrightarrow{u'} \preceq^*_\beta T',\ \#(\Delta_{p'}) = \#(\overrightarrow{p'})$,
  $\mathsf{Ind}(I[\Delta_p] : A := \overrightarrow{U}),\ \mathsf{Ind}(I'[\Delta_{p'}] : A' := \overrightarrow{U'}) \in \Sigma$,
  $\Gamma \vdash_\beta \Pi\Delta_p. U_i \mathbin{@} \overrightarrow{p}\,\overrightarrow{a} \rhd I\ \overrightarrow{p}\ \overrightarrow{u},\ \Gamma \vdash_\beta \Pi\Delta_{p'}. U'_i \mathbin{@} \overrightarrow{p'}\,\overrightarrow{a'} \rhd I'\ \overrightarrow{p'}\ \overrightarrow{u'}$.
  Since t is a constructor we have $T \approx_\beta T'$ so $I\ \overrightarrow{p}\ \overrightarrow{u} \approx_\beta I'\ \overrightarrow{p'}\ \overrightarrow{u'}$. We deduce $I = I'$ and $\overrightarrow{p}\,\overrightarrow{u} \simeq_\beta \overrightarrow{p'}\,\overrightarrow{u'}$. Since Σ contains a unique declaration for each inductive, declarations of I and I' are equal so $\#(\overrightarrow{p}) = \#(\Delta_p) = \#(\Delta_{p'}) = \#(\overrightarrow{p'})$ and we have $\overrightarrow{p} \simeq_\beta \overrightarrow{p'}$. But both vectors are in normal form so they are equal (by reducibility). Then we apply induction hypothesis on the first statement to prove that $\overrightarrow{p}\,\overrightarrow{a} = \overrightarrow{p'}\,\overrightarrow{a'}$ and conclude.

– Case of $t = \langle P\rangle\mathsf{case}\ M\ \mathsf{of}\ \overrightarrow{\overrightarrow{x} \Rightarrow b}$ we have:
$t' = \langle P'\rangle\mathsf{case}\ M'\ \mathsf{of}\ \overrightarrow{\overrightarrow{x} \Rightarrow b'}$, $P =_\epsilon P'$, $M =_\epsilon M'$ and for all i, $b_i =_\epsilon b'_i$
By inversion on $\Gamma \vdash_\beta t : T$ and $\Gamma \vdash_\beta t' : T'$ we have:
$\mathsf{Ind}(I[\Delta_p] : \Pi\Delta_a.\, s := \overrightarrow{\Pi\Delta_i.\, I\ \mathrm{Dom}(\Delta_p)\ \overrightarrow{t_i}}) \in \Sigma$
$\mathsf{Ind}(I'[\Delta_{p'}] : \Pi\Delta_{a'}.\, s' := \overrightarrow{\Pi\Delta'_i.\, I'\ \mathrm{Dom}(\Delta_{p'})\ \overrightarrow{t'_i}}) \in \Sigma$
$\Gamma \vdash_\beta M : I\ \overrightarrow{p}\ \overrightarrow{a} \qquad \Gamma \vdash_\beta M' : I'\ \overrightarrow{p'}\ \overrightarrow{a'}$
$\Gamma \vdash_\beta P : \Pi\Delta_a.\, \Pi x{:}I\ \overrightarrow{p}\ \mathrm{Dom}(\Delta_a).\, s_P = T_P$
$\Gamma \vdash_\beta P' : \Pi\Delta_{a'}.\, \Pi x{:}I'\ \overrightarrow{p'}\ \mathrm{Dom}(\Delta_{a'}).\, s_{P'} = T_{P'}$
$\forall i,\ \Gamma(\Delta_i\sigma_p) \vdash_\beta b_i : P\ (\overrightarrow{t}_i\sigma_p)\ \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{x_i})$
$\forall i,\ \Gamma(\Delta'_i\sigma_{p'}) \vdash_\beta b'_i : P'\ (\overrightarrow{t'}_i\sigma_{p'})\ \mathsf{C}^i_{I'}(\overrightarrow{p'}, \overrightarrow{x_i})$
where $\sigma_k = \{\mathrm{Dom}(\Delta_k) \leftarrow \overrightarrow{k}\}$.
M is not a abstraction (t is well-typed) and not a constructor (t is a normal form), so we can apply the induction hypothesis on M and M' to get $M = M'$. Uniqueness of type yields $I\ \overrightarrow{p}\ \overrightarrow{a} \approx_\beta I'\ \overrightarrow{p'}\ \overrightarrow{a'}$ so $I = I'$ and $\overrightarrow{p}\,\overrightarrow{a} \simeq_\beta \overrightarrow{p'}\,\overrightarrow{a'}$. Since Σ contains a unique declaration for each inductive, the declaration of I and I' are equal. Since $\#(\overrightarrow{p}) = \#(\Delta_p) = \#(\overrightarrow{p'})$ we have $\overrightarrow{p} \simeq_\beta \overrightarrow{p'}$. We can now prove $T_P \approx_\beta T_{P'}$ and apply the induction hypotheses on P and P' to prove $P = P'$. Finally we need to prove equality of each branch. Since $I = I'$ and $P = P'$ we have

$$\Gamma(\Delta_i\sigma_{p'}) \vdash_\beta b'_i : P\ (\overrightarrow{t}_i\sigma_{p'})\ \mathsf{C}^i_I(\overrightarrow{p'}, \overrightarrow{x_i})$$

Since $\overrightarrow{p} \simeq_\beta \overrightarrow{p'}$, context conversion yields

$$\Gamma(\Delta_i\sigma_p) \vdash_\beta b'_i : P\ (\overrightarrow{t}_i\sigma_{p'})\ \mathsf{C}^i_I(\overrightarrow{p'}, \overrightarrow{x_i})$$

and we have

$$P\ (\overrightarrow{t}_i\sigma_{p'})\ \mathsf{C}^i_I(\overrightarrow{p'}, \overrightarrow{x_i}) \simeq_\beta P\ (\overrightarrow{t}_i\sigma_p)\ \mathsf{C}^i_I(\overrightarrow{p}, \overrightarrow{x_i})$$

so we can apply the inductive hypotheses to prove $b_i = b'_i$, which concludes the proof.