Normalized Types

Pierre Courtieu

Université Paris-Sud, Laboratoire de recherche en Informatique (LRI),
Bâtiment 490 – 91405 Orsay Cedex, France –
http://www.lri.Fr/

Abstract. We present a new method to specify a certain class of quotient in intentional type theory, and in the calculus of inductive constructions in particular. We define the notion of "normalized types". The main idea is to associate a *normalization function* to a type, instead of the usual relation. This function allows to compute on a particular element for each equivalence class, avoiding the difficult task of computing on equivalence classes themselves. We restrict ourselves to quotients that allow the construction of such a function, i.e. quotient having a canonical member for each equivalence class. This method is described as an extension of the calculus of constructions allowing normalized types. We prove that this calculus has the properties of strong normalization, subject reduction, decidability of typing. In order to show the example of the definition of $\mathbb{Z}$ by a normalized type, we finally present a pseudo Coq session.

1 Introduction

1.1 The calculus of inductive constructions

Type theory is a fruitful formalism for automated proof systems like Coq, Lego, Agda/alfa etc. The expressive power of this framework is comparable to set theory. However it appeared that the definition of complex structures is easier and computationally more powerful if the type theory is enriched with some constructions. In particular *the calculus of inductive constructions* ($\mathcal{CIC}$) [14] is an extension of the calculus of construction [6] where it is possible to define new types by giving the list of its typed constructors, in a sort of ML-style. For example :

Inductive Nat := *0:Nat* | *S:Nat* → *Nat.*
Inductive AList := *nil: (A:Set) (AList A)*
| *cons:(A:Set) A* → *(AList A)* → *(AList A).*
Inductive BListn := *nil: (BListn 0)*
| *cons: (n:Nat) Bool* → *(BListn n)* → *(BListn (S n)).*

By definition the terms of an *inductive* type T is *the least set of terms* recursively built from its constructors. This is stated by the induction schemes, also called *elimination principles*, like for example the one on natural: $\forall P\ \{(P\ 0) \wedge (\forall x(P\ x) \Rightarrow (P\ (S\ x)))\} \Rightarrow \forall x(P\ x)$. It is possible to define functions on an inductive type by pattern matching in ML-style, that can be evaluated in the usual way. For example:

$$parity = \lambda x{:}Nat.\ \ Case\ x\ of\ 0 \Rightarrow true\ /\ (S\ n) \Rightarrow (not\ (parity\ n))\ end$$

In the calculus of inductive constructions, these notions are stated by typed terms, in the *Curry-Howard* spirit. The elimination principle above is for example expressed by the term:

$$Nat\_rec{:}(P{:}Nat{\rightarrow}\mathsf{Set})(P\ 0){\rightarrow}((m{:}Nat)(P\ m){\rightarrow}(P\ (S\ m))){\rightarrow}(m{:}Nat)(P\ m)$$

where an expression of the form $(x{:}A)T$ (also written $\Pi x : A.T$) is the notation for the dependent product, abbreviated in $A \rightarrow T$ when T does not depend on x.

Using *Nat_rec*, we can define (possibly recursive) functions on *Nat* by pattern matching on constructors. We can also make proofs by cases or by induction on *Nat*. We can deduce from *Nat_rec* a non dependent principle, easier to use when the type P is not dependent:

$$Nat\_rec'{:}\ (P{:}\mathsf{Set})\ P{\rightarrow}(Nat{\rightarrow}P{\rightarrow}P){\rightarrow}Nat{\rightarrow}P$$

The function parity is written in the $\mathcal{CIC}$ using *Nat_rec'* :

$$parity = (Nat\_rec'\ bool\ true\ ([y{:}nat][b{:}bool](not\ b)))$$

where *[x:A]t* stands for $\lambda x : A.t$. In order to evaluate such functions in the $\mathcal{CIC}$, new rewriting rules are added for the recursors:

$$\begin{array}{ll} (Nat\_rec'\ T\ t_0\ t_1\ 0) & \rightarrow_\iota t_0 \\ (Nat\_rec'\ T\ t_0\ t_1\ (S\ x)) & \rightarrow_\iota (t_1\ x\ (Nat\_rec'\ T\ t_0\ t_1\ x)) \end{array}$$

where t_0 and t_1 are two terms of type T and $nat{\rightarrow}T{\rightarrow}T$ respectively. Notice how structural recursion is simulated and how the head constructor of the last argument (*0* or S) is used to choose which *branch* of the recursor must be used.

In the version of the $\mathcal{CIC}$ that we consider, this new reduction, called ι-reduction is part of the internal reduction of the system, as for β (evaluation of applications). We do not consider η (evaluation of dummy abstractions) in our work. More precisely, the internal reduction $\longrightarrow_{\mathcal{CIC}}$ which defines the evaluation mechanism on terms, is :

$$\longrightarrow_{\mathcal{CIC}}\ = \longrightarrow_\iota\ \cup\ \longrightarrow_\beta$$

From the theorem prover perspective, it is very important that the evaluation mechanism terminates. So in $\mathcal{CIC}$ restrictions on the definition of functions are made to allow only terminating functions to be defined (see appendix B, and [14]). This evaluation mechanism is closely related to the internal notion of equality of $\mathcal{CIC}$, as we explain in the next section.

1.2 The notion of equality

The notion of equality in type theory is a delicate problem. The undecidability in general of the usual notion of equality in mathematics makes impossible any implementation of a general decision procedure. However, a lot of work has been done to find weaker decidable equivalence relations between terms of the type theory, in order to:

- Make decision procedures to minimize the number of proofs of equality to be made by hand.

– Make typing decidable in presence of a *conversion rule* of the form (needed in a dependently typed framework):

$$\textbf{Conv} : \frac{\Gamma \vdash t : T_1 \quad T_1 \equiv T_2}{\Gamma \vdash t : T_2}$$

where $\equiv$ is an equivalence relation between terms, Γ is a typing environment, and t, T_1 and T_2 are terms. We see here that in order to keep a decidable typing procedure, $\equiv$ must remain decidable.

A decidable typing is not an absolute requirement, systems like PVS do not have a decidable typing. However, there are evident practical advantages in decidability of typing and internal equality and we will consider it necessary in our paper according to the point of view that prevailed in the conception of Coq.

Since the classical (undecidable) notion of equality is still necessary, the $\mathcal{CIC}$, as other systems of type theory, uses several notions of equality. In $\mathcal{CIC}$ we first have a notion of *internal equality*, also called definitional equality[1]. This equality is a congruence on terms (and types because we are in a type dependent framework) and is the equality used in the *conversion rule* of the type system. More precisely, the internal equality $\equiv_{\mathcal{CIC}}$ of $\mathcal{CIC}$ is the reflexive, symmetric and transitive closure of $\longrightarrow_{\mathcal{CIC}}$.

We have a second notion of equality, called *propositional equality* ($\stackrel{p}{\equiv}_{\mathcal{CIC}}$) also called *Leibniz equality*. It is defined in the calculus and can be used and extended by the user. It will not be considered in the typing rules and is not necessarily decidable. It is defined such that $\equiv_{\mathcal{CIC}} \subseteq \stackrel{p}{\equiv}_{\mathcal{CIC}}$, thus we will call *user equality*, noted $\stackrel{u}{\equiv}_{\mathcal{CIC}}$, the minimal relation such that: $\stackrel{p}{\equiv}_{\mathcal{CIC}} = \equiv_{\mathcal{CIC}} \cup \stackrel{u}{\equiv}_{\mathcal{CIC}}$ that is the part of $\stackrel{p}{\equiv}_{\mathcal{CIC}}$ that is not in $\equiv_{\mathcal{CIC}}$.

Notice here that nothing prevents the user from generating an inconsistency, for example by defining P such that $P(True, False)$. We see in the next section that the fact that the propositional equality is not restricted to $\equiv_{\mathcal{CIC}}$ creates more subtle problems.

To sum up this section, we can say that $\stackrel{p}{\equiv}_{\mathcal{CIC}}$ is the mathematical equality and $\equiv_{\mathcal{CIC}}$ is the sub-relation of $\stackrel{p}{\equiv}_{\mathcal{CIC}}$ that is considered (and decided) during typing. Of course extending $\equiv_{\mathcal{CIC}}$ is interesting as more terms will be identified internally.

1.3 The problem of non free structures

As we saw in section 1.1 the ι rule only checks the head constructor of the argument of a function to decide which reduction rule to apply. This mechanism

[1] In fact, Martin-Löf in its type theory gives four notions of equality: intentional (definitional) equality, judgment equality, type equality, and propositional equality. In the $\mathcal{CIC}$ these categories are not exactly relevant as intentional equality is much more powerful than in Martin-löf's theory, and judgment and type equalities are somehow replaced by internal or propositional equality.

works well when the structures defined by inductive types are *free*, which means that two terms starting with two different constructors cannot be equal (even for $\overset{u}{\equiv}_{\mathcal{CIC}}$). But it fails in presence of equations between head constructor terms[2] (in $\overset{u}{\equiv}_{\mathcal{CIC}}$). Indeed, in this case, two equal terms (by $\overset{p}{\equiv}_{\mathcal{CIC}}$)$t_1$ and t_2, starting by two different constructors, can generate an incoherence by $\longrightarrow_{\mathcal{CIC}}$. Suppose for example that we want to define the natural numbers modulo 2, we can naively state the axiom *Eq0mod2:* $0 \overset{u}{\equiv}_{\mathcal{CIC}} (S\ (S\ 0))$ but then consistency of the system is compromised, as shown by the simple function f[3]:

$$f = \lambda x{:}Nat.\ \ Case\ x\ of \quad 0 \Rightarrow\ True \\ \qquad\qquad\qquad\qquad\qquad\qquad |\ (S\ n) \Rightarrow\ False\ end$$

generating two equalities $(f\ 0) \equiv_{\mathcal{CIC}}\ True$ and $(f\ 0) \overset{u}{\equiv}_{\mathcal{CIC}} (f\ (S\ S\ 0)) \equiv_{\mathcal{CIC}}\ False$ leading trivially to $True \overset{p}{\equiv}_{\mathcal{CIC}}\ False$.

For this reason, inductive types allow only to specify free structures. This means that it is not possible to define quotients on inductive types in a simple way in systems like `Coq`.

However, mathematical structures like integers ($\mathbb{Z}$), integers modulo ($\mathbb{Z}/n\mathbb{Z}$), rationals ($\mathbb{Q}$), or Sets are intrinsically quotients, that we want to define from an underlying type and an equivalence on this type.

Our contribution, the *normalized types* is a method to specify structures where $\overset{in}{\equiv}$ is safely extended by the user, allowing to define a certain class of quotient. It has been inspired by two existing methods, the *quotient types* [9], [11], [5], and the *congruence types* [2], that we both briefly present in the following.

1.4 Related work

The definition of non free structures in intentional type theory has been studied by Backhouse, Hofmann, Barthes, Geuvers, Jacobs and recently by S. Boutin. Roughly two methods were proposed: quotient types and congruence types, both inspired by previous works on extensional type theory.

Quotient types as presented in [5] are an axiomatization of quotients from the set theory, that has been implemented in the `Coq` system. A quotient type T/R is built from a type T (that we shall call the *underlying type*), and a relation R on this type. For all $x : T$ we define the term $(In\ x) : T/R$. A set of axioms defines the properties of T/R according with the classical notion of quotient, for example: $\forall x, y : T.(R\ x\ y) \rightarrow (In\ x) \equiv_{user} (In\ y)$ that defines the equality among the elements of the quotient. This axiomatization, by taking a set theoretical approach, is very general, but because of its axiomatic nature, does not catch the computational aspect of quotients (In particular because of the use of $\equiv_{user}$ above instead of $\equiv_{\mathcal{CIC}}$).

[2] Terms whose head symbol is a constructor.

[3] Notice that strictly speaking we use here a strong version of `Nat_rec'`.

Martin Hofmann, in [10] and [9] already defines quotients this way. By extending the Martin-Löf type theory with quotient types and by giving an interpretation of it in the pure Calculus of Constructions, he gives good properties to quotient types. Our present work follows a similar method.

A common aspect of these works is that the elimination principles for quotient is split in two principles. The first is weak and allows to define functions on the quotient from functions on the underlying type. This *lifting* operation is allowed when the function is *compatible* with the relation of the quotient, according to the classical set theory mechanism. The second is strong and allows to make inductions on quotient types, again by lifting proofs made on the underlying type. There is no compatibility condition for this principle. But it is clear that in some cases it is possible to define better induction schemes.

Congruence types are a generalization of inductive types which could be called "inductive types with relation" ([13] [5]), or "inductive types with rewriting" [2]. This last point of view is the closest of our approach. It consists in the association of an inductive type $\underline{T}$ and a canonical (i.e. confluent and terminating) term rewriting system(TRS) ρ, which will be added to the internal reduction and equality of the system. This defines a new type T. Despite not being an inductive type, T has a good computational behavior, because we can use ρ to link any closed term of $\underline{T}$ to a unique term in T : its normal form by ρ. This method allows a satisfying representation of quotients when the relation R can be oriented in a canonical TRS. In particular better induction schemes can be proved by hand using a notion of *fundamental constructor*. However, adding rewriting systems to internal reduction dynamically leads to the difficult problems of termination criteria and interaction between rewriting systems [3].

1.5 Plan

We give a general description of normalized types in the next section. Then we define formally our extension of the calculus of constructions in section 2. We prove properties of this system in section 3. Finally we conclude with some considerations on our system and with some ideas of further work in 4. In appendix A the example of the definition of $\mathbb{Z}$ in a pseudo Coq session is developed. We give in the appendix B a short definition of the calculus of construction as defined in [14].

2 The Calculus of Inductive Constructions with normalized types

Let A be a type and nf a function from A to A, we define a new type $\mathsf{Norm}(A, nf)$ called "type A normalized by nf". Its elements are, and are only, of the form: $\mathsf{Class}(A, nf, t)$, where t is a term of type A. This will expressed by the elimination principle. The main idea of this work is to make $\mathsf{Class}(A, nf, t)$ and $\mathsf{Class}(A, nf, u)$

equivalent for internal equality (convertible) if $(nf\ t)$ and $(nf\ u)$ are equivalent. Reduction is defined to avoid the coherency problems cited in section 1.3.

Our system is a variant of quotient types in the spirit of congruence types of Barthe and Geuvers. We take advantages of both methods:

- We use the idea of the association of a type with a computational object. Barthe uses a TRS, we will use a normalization function nf, i.e. a term of the original system, avoiding the problem of mixing reductions cited previously.
- We will define a new calculus extending the calculus of inductive constructions. We use an interpretation from our calculus into $\mathcal{CIC}$ to define our notion of internal equality.

By a slight addition to the reduction rules we can add the normalized types to the Calculus of Inductive Constructions. The modification mainly consists in the enrichment of the conversion and reduction rules in order to make $\mathsf{Class}(A, nf, t)$ and $\mathsf{Class}(A, nf, u)$ convertible provided that t and u have the same canonical form. The subject of the end of this paper is the formal definition and the study of the main properties of this extension.

We use a method similar to [1] but in the context of inductive types, we first extend the syntax, typing and reduction rules of $\mathcal{CIC}$ and then we prove the properties of the new calculus ($\mathcal{CIC}^{nf}$) using a translation from $\mathcal{CIC}^{nf}$ to $\mathcal{CIC}$ that has some strong properties.

2.1 Syntax

The syntax of $\mathcal{CIC}^{nf}$ is based on the notations of B. Werner's thesis [14] where a precise description of the $\mathcal{CIC}$ can be found (a short one can be found in appendix B.

We take here the following hierarchy of sorts: Set:Type:Extern.

Variables : $V ::= x, y, z \ldots$

Sorts: $S ::= \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Extern}$

Terms (all terms of $\mathcal{CIC}$ belong also to $\mathcal{CIC}^{nf}$):

$$\begin{aligned} T ::= {} & V \mid S \mid [V:T]T \mid (V:T)T \mid TT \\ & \mid \mathsf{Ind}(V:T)(\boldsymbol{T}) \mid \mathsf{Constr}(n,T)\ n \in \mathbb{N} \mid \mathsf{Elim}(T,T,\boldsymbol{T},T)\{\boldsymbol{T}\} \\ & \mid \mathsf{Norm}(T,T) \mid \mathsf{Class}(T,T,T) \mid \mathsf{Elimnorm}(T,T,T,T) \end{aligned}$$

$\boldsymbol{T}$ denotes a sequence of terms. Ind, Constr and Elim are the usual constructions for inductive types. Norm, Class and Elimnorm are respectively the type constructor, term constructor and destructor for normalized types.

We will use the usual notation $A \to B$ in place of $(x : A)B$ when B does not depend on x. To increase readability, we will often use Coq-like notations for pattern matching expressions:

$$\textit{Cases } t \textit{ of } \langle pattern1\rangle \ \Rightarrow u_1 \mid \langle pattern2\rangle \Rightarrow u_2 \ldots \textit{end.}$$

in place of the corresponding $\mathsf{Elim}(t_1, t_2, t_3, t)\ \{\boldsymbol{u_i}\}$, hiding arguments t_1, t_2 and t_3 that can be deduced from context.

We will as usual note $t[x \leftarrow u]$ the term t where all free occurrences of x have been replaced by u. The notion of free variable extends well to our new constructions, and all usual properties of substitutions are preserved.

Finally, we define typing environments as sets of pairs of the form $(x : T)$. As usual $[]$ will denote the empty environment, and $\Gamma :: x : T$ the environment $\Gamma \cup \{(x : T)\}$. We will omit parenthesis when it is not ambiguous.

2.2 Computation

Definition 1. *Let $\longrightarrow_{nf}$ be the following rewriting rule:*

$$\mathsf{Elimnorm}(A, nf, f, \mathsf{Class}(A, nf, t)) \longrightarrow_{nf} (f\ (nf\ t))$$

the reduction $\longrightarrow_{\mathcal{CIC}^{nf}}$ of $\mathcal{CIC}^{nf}$ is the congruent closure of $\longrightarrow_{nf} \cup \longrightarrow_{\iota} \cup \longrightarrow_{\beta}$

Remark 1. Let t_1 and t_2 be two terms of $\mathcal{CIC}$ such that $t_1 \longrightarrow_{\iota\beta} t_2$ in $\mathcal{CIC}$, then we have also $t_1 \longrightarrow_{\iota\beta} t_2$ in $\mathcal{CIC}^{nf}$. $\longrightarrow_{\mathcal{CIC}^{nf}}$ reduction preserves typing.

2.3 Conversion and internal equality

Conversion is defined using a combination of an interpretation φ of terms of $\mathcal{CIC}^{nf}$ into other terms of $\mathcal{CIC}^{nf}$, and a new reduction $\longrightarrow_{\iota\beta+nf'}$ applied to the interpreted terms. This unusual definition is necessary since we want two terms to be convertible when there canonical forms (calculated by the interpretation) are equivalent for a certain relation (the new notion of reduction).

Since φ is not necessarily idempotent, it is impossible to define the conversion as the closure of a reduction. We see here that deciding equality ($\equiv_{nf+\mathcal{CIC}}$) and computing ($\longrightarrow_{\mathcal{CIC}^{nf}}$) are not anymore the same issues, but the two notions have to be compatible, as it is stated in property 1.

Definition 2. *We define φ on terms and environments recursively as follows:*

- $\varphi(\mathsf{Class}(A, nf, t)) = \mathsf{Class}(\varphi(A), \varphi(nf), (\varphi(nf)\ \varphi(t)))$,
- $C[\boldsymbol{t}] = C[\varphi(\boldsymbol{t})]$ *where C is a context different than* $\mathsf{Class}(\_)$.
- $\varphi([]) = []$ *and* $\varphi(\Gamma :: x : T) = \varphi(\Gamma) :: x : \varphi(T)$

Definition 3. *We define $\longrightarrow_{nf'}$, $\equiv_{\iota\beta+nf'}$, $\equiv_{\mathcal{CIC}^{nf}}$ as follows:*

- $\mathsf{Elimnorm}(A, nf, f, \mathsf{Class}(A, nf, t)) \longrightarrow_{nf'} (f\ t)$
- $\equiv_{\iota\beta+nf'}$ *is the congruent closure of $\longrightarrow_{\iota}$, $\longrightarrow_{\beta}$ and $\longrightarrow_{nf'}$.*
- t_1 *and* t_2 *are convertible ($t_1 \equiv_{\mathcal{CIC}^{nf}} t_2$) iff $\varphi(t_1) \equiv_{\iota\beta+nf'} \varphi(t_2)$.*

Property 1. If $t \longrightarrow_{\mathcal{CIC}^{nf}} u$ then $t \equiv_{\mathcal{CIC}^{nf}} u$.

2.4 Typing

The typing system contains the rules of $\mathcal{CIC}$ (given in appendix B), except the conversion rule, plus the following rules:

$$\textbf{FormN} : \frac{\Gamma \vdash A : \mathsf{Set} \quad \Gamma \vdash nf : A \to A}{\Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}} \quad \textbf{IntroN} : \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}}{\Gamma \vdash \mathsf{Class}(A, nf, t) : \mathsf{Norm}(A, nf)}$$

$$\textbf{Conv} : \frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_1, T_2 : s \quad T_1 \equiv_{\mathcal{CIC}^{nf}} T_2}{\Gamma \vdash t : T_2}$$

$$\mathbf{ElimN_{nodep}} : \frac{\Gamma \vdash P : Sort \quad \Gamma \vdash t : \mathsf{Norm}(A, nf) \qquad \Gamma \vdash H : A \rightarrow P}{\Gamma \vdash \mathsf{Elimnorm}(A, nf, H, t) : P}$$

$$\mathbf{ElimN} : \frac{\Gamma \vdash P : \mathsf{Norm}(A, nf) \rightarrow Sort \quad \Gamma \vdash t : \mathsf{Norm}(A, nf) \qquad \Gamma \vdash H : (s : A)(P\ \mathsf{Class}(A, nf, s))}{\Gamma \vdash \mathsf{Elimnorm}(A, nf, H, t) : (P\ (IdNorm(A, nf, t)))}$$

with $IdNorm(\mathrm{A},nf,\mathrm{t}) =_{def} \mathsf{Elimnorm}(\mathrm{A},nf,([\mathrm{x{:}A}]\ \mathsf{Class}(\mathrm{A},nf,\mathrm{x})),\mathrm{t})$

In the rule **ElimN** we use the term $IdNorm$, which maps $\mathsf{Class}(\ldots, x)$ to $\mathsf{Class}(\ldots, (nf\ \ x))$. It is a consequence of the implicit normalization done with the reduction $\longrightarrow_{nf}$ (and φ) defined previously. It is necessary to ensure that the reduction $\longrightarrow_{nf}$ preserves typing. We see here that $\mathsf{Elimnorm}(A, nf, H, t)$ is not the proof that P is verified by t but by the canonical form of t.

Now it is easy to replace the property $(nf\ s\ \overset{p}{\equiv}_{\mathcal{CIC}}\ s)$ by any equivalent inductive predicate to have a powerful principle to define function or make inductive proofs on normalized types. See appendix A for an example.

3 Properties of $\mathcal{CIC}^{nf}$

Important results about this system are:

1. *subject reduction*, which proof is classical, similar to what can be found in [14] or [8].
2. *strong normalization* $\longrightarrow_{\mathcal{CIC}^{nf}}$ on well typed terms, that is proved in the following sections as a consequence of the analog property of $\mathcal{CIC}$.
3. *decidability of typing*, that is a consequence of the decidability of $\equiv_{\mathcal{CIC}^{nf}}$, that is a consequence of 4 and 5
4. *strong normalization* of $\longrightarrow_{\iota\beta+nf'}$ on well typed terms, which proof is completely similar to 2.
5. *confluence (Church-Rosser)* of $\longrightarrow_{\iota\beta+nf'}$, proved using a classical the notion of *parallel reduction.*

3.1 A translation from $\mathcal{CIC}^{nf}$ to $\mathcal{CIC}$

To prove strong normalization of our reduction $\longrightarrow_{\iota\beta+nf}$ we use the translation $\langle\rangle$ from $\mathcal{CIC}^{nf}$ to $\mathcal{CIC}$, such that if $t \longrightarrow_{\mathcal{CIC}^{nf}} t'$, then $\langle t\rangle \longrightarrow_{\mathcal{CIC}} \langle t'\rangle$. Thus, since $\langle\rangle$ preserves also typing, $\mathcal{SN}(\mathcal{CIC}^{nf})$ is a consequence of $\mathcal{SN}(\mathcal{CIC})$, which is well known.

Definition 4. *$\langle\rangle$ is defined by induction as follows:*

$$\begin{array}{lcr}
\langle \mathit{Norm}(A, nf)\rangle & = & (Indnorm\ \langle A\rangle\ \langle nf\rangle) \\
\langle \mathit{Class}(A, nf, t)\rangle & = & (indclass\ \langle A\rangle\ \langle nf\rangle\ \langle (nf\ t)\rangle) \\
\langle \mathit{Elimnorm}(A, nf, f, t)\rangle & = & (\langle f\rangle\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)) \\
\langle C[\boldsymbol{t}]\rangle & = & C[\langle \boldsymbol{t}\rangle] \\
 & & \text{For any other construction } C \\
\multicolumn{3}{c}{\text{We extend trivially } \langle\rangle \text{ to environments:}} \\
\langle []\rangle & = & [] \\
\langle \Gamma :: x : T\rangle & = & \langle \Gamma\rangle :: x : \langle T\rangle
\end{array}$$

Notice the interpretation of the new terms of $\mathcal{CIC}^{nf}$ into an inductive type (*Indnorm* defined below) of $\mathcal{CIC}$. Terms of the form $\mathsf{Class}(...)$ are normalized by $\langle\rangle$ via nf. The type and terms of $\mathcal{CIC}$used in the translation are defined as follows:

- *Indnorm* is a parameterized inductive type, it corresponds to the type Norm:
  Indnorm := [A:Set] [nf:A $\rightarrow$ A] Ind (X:Set) {A $\rightarrow$ X}
- the unique constructor of *Indnorm* corresponds to the construction Class:
  indclass:= [A:Set] [nf:A $\rightarrow$ A] Constr(1,(Indnorm A nf)) [4]
- the destructor *rep* of *Indnorm* corresponds to $\mathsf{Elimnorm}$:
  rep:=[A:Set][nf:A$\rightarrow$A][t:(Indnorm A nf)]
  Cases t of (indclass A nf x)$\Rightarrow$x end.

  The type of *rep* is the following: *(A:Set) (nf:A $\rightarrow$ A) (Indnorm A nf) $\rightarrow$ A.*

3.2 Properties of the translation $\langle\rangle$

In this section we prove properties of $\langle\rangle$ that will allow us to state in the following section the strong normalization, subject reduction and confluence modulo conversion of $\mathcal{CIC}^{nf}$. This is the technical part. The two main properties of $\langle\rangle$ are that it preserves the typing relation and reduction.

We will note $t \stackrel{\langle\rangle}{=} t'$ when t is equal to t' by definition of $\langle\rangle$.

In order to prove strong normalization of $\mathcal{CIC}^{nf}$ from strong normalization of $\mathcal{CIC}$, $\langle\rangle$ needs to preserve the typing relation. Before proving this property we state a small lemma:

Lemma 1. *For all terms X, Y and T, and for all environment Γ, if $\Gamma \vdash_{\mathcal{CIC}}$ $(Indnorm\ X\ Y) : T$ or $\Gamma \vdash_{\mathcal{CIC}} T : (Indnorm\ X\ Y)$, then $\Gamma \vdash_{\mathcal{CIC}} X : \mathsf{Set}$ and $\Gamma \vdash_{\mathcal{CIC}} Y : X \rightarrow X$.*

Proof. This is deduced from the typing rules and the type of *Indnorm* which is *(A:Set) (nf:A $\rightarrow$ A) Set.*

Lemma 2. *If $\Gamma \vdash_{\mathcal{CIC}^{nf}} u : T$ then $\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle u\rangle : \langle T\rangle$.*

Proof. By induction on the proof that $\Gamma \vdash_{\mathcal{CIC}^{nf}} u : T$. All cases from $\mathcal{CIC}$ are immediate since the typing rules of $\mathcal{CIC}$ are also typing rules of $\mathcal{CIC}^{nf}$. Finally the only difficulty is the case of the rule **ElimN**:

- Last rule used is **ElimN**, we know that:

$$u = \mathsf{Elimnorm}(A, nf, f, t), \text{ and thus } \langle u\rangle \stackrel{\langle\rangle}{=} (\langle f\rangle\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle))$$
$$\Gamma \vdash_{\mathcal{CIC}^{nf}} \mathsf{Elimnorm}(A, nf, f, t) : (P\ (IdNorm(A, nf, t)))$$

From the rule **ElimN** we know that the following assertions hold:

[4] To make things clear, we can say that this is what is defined in Coq when we write the following definition:
Inductive Indnorm [A:Set, nf:A $\rightarrow$ A]:= indclass : A $\rightarrow$ (Indnorm A nf).
Where A and *nf* are *parameters* of the inductive type *Indnorm*.

(i) $\Gamma \vdash_{\mathcal{CIC}^{nf}} f : (s : A)(P\ \mathsf{Class}(A, nf, s))$
So by ind. hyp.: $\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle f\rangle : (s : \langle A\rangle)(\langle P\rangle\ (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ s)))$
(ii) $\Gamma \vdash_{\mathcal{CIC}^{nf}} t : \mathsf{Norm}(A, nf)$
So by ind. hyp.: $\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle t\rangle : (\mathit{Indnorm}\ \langle A\rangle\ \langle nf\rangle)$
(iii) $\Gamma \vdash_{\mathcal{CIC}^{nf}} P : \mathsf{Norm}(A, nf) \to \mathit{Sort}$.

By lemma 1 and (ii), we have:
$\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle A\rangle : \mathsf{Set}$, and $\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle nf\rangle : \langle A\rangle \to \langle A\rangle$
and so, from the type of *rep* we have:
$\langle\Gamma\rangle \vdash_{\mathcal{CIC}} (\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle) : \langle A\rangle$
let us denote the term $(\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)$ by v. By applying the typing rule for application, we can conclude that:
$\langle\Gamma\rangle \vdash_{\mathcal{CIC}} (\langle f\rangle\ v) : (\langle P\rangle\ (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ s)))[s \leftarrow v]$.
Therefore:

$$\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle u\rangle : (\langle P\rangle\ (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ (\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)))) \qquad (1)$$

On the other hand, we can verify that:
$\langle P\ \ (\mathit{IdNorm}(A, nf, t))\rangle \equiv_{\mathcal{CIC}^{nf}} (\langle P\rangle\ \ (\mathit{indclass}\ \ \langle A\rangle\ \ \langle nf\rangle\ \ (\mathit{rep}\ \ \langle A\rangle\ \ \langle nf\rangle\ \ \langle t\rangle)))$
which is the type of $\langle u\rangle$ in $\langle\Gamma\rangle$(Cf. (1)). So by the conversion rule of $\mathcal{CIC}$ we conclude that:
$\langle\Gamma\rangle \vdash_{\mathcal{CIC}} \langle u\rangle : \langle P\ \ (\mathit{IdNorm}(A, nf, t))\rangle$.

The second property that we want in order to prove that $\mathcal{CIC}^{nf}$ is normalizing is that $\langle\rangle$ preserves reductions (lemma 4). We need to prove first that $\langle\rangle$ is coherent with substitutions, which is rather immediate.

Lemma 3. *If $t = u[x \leftarrow v]$ is a term of $\mathcal{CIC}^{nf}$, then $\langle t\rangle = \langle u\rangle[x \leftarrow \langle v\rangle]$.*

Proof. By induction on u and then by cases. We suppose without loss of generality that all occurrences of x in u are free.

Now we can state the preservation of reductions:

Lemma 4. *For all well typed terms t_1 and t_2 of $\mathcal{CIC}^{nf}$, if $t_1 \to_{\iota\beta+nf} t_2$ then $\langle t_1\rangle \to^{+}_{\iota\beta} \langle t_2\rangle$.*

Proof. By induction on t_1.

- Base case: $t_1 = x$, Extern, Set or Type, then t_1 is not reducible by $\to_{\iota\beta+nf}$.
- $t_1 = (x : u)v$, then $\langle t_1\rangle = (x : \langle u\rangle)\langle v\rangle$ the reduction is necessarily done on a strict sub-term of t_1. There are two cases:
  1. $t_2 = (x : u')v$ with $u \to_{\iota\beta+nf} u'$, then $\langle t_2\rangle = (x : \langle u'\rangle)\langle v\rangle$. By induction hypothesis, $\langle u\rangle \to^{+}_{\iota\beta} \langle u'\rangle$, thus $\langle t_1\rangle = (x : \langle u\rangle)\langle v\rangle \to^{+}_{\iota\beta} (x : \langle u'\rangle)\langle v\rangle = \langle t_2\rangle$. OK.
  2. or $t_2 = (x : u)v'$ with $v \to_{\iota\beta+nf} v'$, same argument. OK.
- The following case are similar to the previous case: $t_1 = [x : u]v$, $t_1 = \mathsf{Ind}(x : u)\boldsymbol{v}$, $t_1 = \mathsf{Constr}(i, u)$, $t_1 = \mathsf{Norm}(u, v, w)$, $t_1 = \mathsf{Class}(t, u, v, w)$.
- $t_1 = (u\ v)$, then $\langle t_1\rangle = (\langle u\rangle\ \langle v\rangle)$, we distinguish two cases:

1. if the reduction is in a strict sub-term of t_1, then the same argument as in previous cases holds again.
2. if the reduction is on the head of t_1, then it is a β-reduction, $t_1 = ([x : T]u'\ v)$, and $t_2 = u'[x \leftarrow v]$. So $\langle t_1 \rangle = ([x : \langle T \rangle]\langle u' \rangle\ \langle v \rangle)$. By lemma 3 $\langle t_2 \rangle = \langle u' \rangle[x \leftarrow \langle v \rangle]$ and therefore $\langle t_1 \rangle \rightarrow_\beta \langle t_2 \rangle$. OK.

– $t_1 = \mathsf{Elim}(u_1, u_2, \boldsymbol{v_i}, u_3)\{\boldsymbol{f_j}\}$, this case is similar to the previous one, giving details here would involve to define precisely ι-reduction, which is rather long.

– $t_1 = \mathsf{Elimnorm}(A, nf, f, t)$, then $\langle t_1 \rangle = (\langle f \rangle\ (rep\ \langle A \rangle\ \langle nf \rangle\ \langle t \rangle))$, there are two cases:
1. if the reduction is in a sub-term of t_1, then the above arguments holds.
2. if the reduction is done on the head of t_1, then it is a nf-reduction, and we know that:
   - $t = \mathsf{Class}(A, nf, u)$ and therefore
     $\langle t_1 \rangle = (\langle f \rangle\ (rep\ \langle A \rangle\ \langle nf \rangle\ \langle \mathsf{Class}(A, nf, u) \rangle))$
     $\stackrel{\langle\rangle}{=} (\langle f \rangle\ (rep\ \langle A \rangle\ \langle nf \rangle\ (indclass\ \langle A \rangle\ \langle nf \rangle\ (\langle nf \rangle\ \langle u \rangle))))$.
     Which can be reduced by β and ι to
     $\rightarrow^+_{\beta i} (\langle f \rangle\ (\langle nf \rangle\ \langle u \rangle))$
   - $t_2 = (f\ (nf\ u))$. and therefore $\langle t_2 \rangle$ is equal to $(\langle f \rangle\ (\langle nf \rangle\ \langle u \rangle))$.

   We have proved that $\langle t_1 \rangle \rightarrow_\beta \langle t_2 \rangle$. OK.

3.3 Strong normalization

Theorem 1. *If there exists an infinite reduction Δ starting from a well typed term t of $\mathcal{CIC}^{nf}$ by $\rightarrow_{\iota\beta+nf}$, then there exists an infinite reduction Δ' starting from a well typed term of $\mathcal{CIC}$ ($\langle t \rangle$) by $\rightarrow_{\iota\beta}$.*

Thus $\mathcal{CIC}^{nf}$ is strongly normalizing on well typed terms.

Proof. The reduction exists by iteration of lemma 4, and $\langle t \rangle$ is well typed by lemma 2.

4 Conclusion

We presented a method to specify a certain class of quotient. Our choice of a function instead of a term rewriting system as in [2] is motivated by the fact that functions are the computational object of the type theory. This allows to use a rather simple extension of $\mathcal{CIC}$ and its reduction. However, defining nf by a rewriting system remains a good idea for several reasons, in particular because it is possible to reduce in a term at any position, which is not possible in general with a recursive function and third it is more efficient.

The class of quotients that we can represent this way is the same as for [2], i.e. quotient whose relation can be "oriented" into a computation.

For Rationals we can define nf as the function that reduces fractions to irreducible fractions, but one needs more work to have a nice definition of $\mathbb{Q}$.

As we said in the beginning of the paper, focusing on a particular member of an equivalence class at the moment of computation can be seen as a weakness of our approach. But the fact that we use this artifice only when *computing* allows to stay at the level of equivalence classes when *reasoning*. Indeed, the terms `Class(...,OI)` and `Class(..., (SI (PI OI)))` are identified only at conversion level (of course they are propositionally equal) but are not reduced one to the other.

Anyway it is clear that an implementation of normalized types should propose several principles as we said previously. It should by the way be interesting to see how automated induction methods as in [4] could be used to generate them automatically. Indeed, such methods are for example able to generate `NormInt` of previous section.

References

[1] G. Barthe. Extensions of pure type systems. In *proc TLCA'95*, volume 902 of *lncs*. Springer-Verlag, 1995.

[2] G. Barthe and H. Geuvers. *Congruence types*. In H. Kleine Buening, editor, *Proc. Conf. Computer Science Logic*, volume 1092 of *Lecture Notes in Computer Science*, pages 36–51, 1995.

[3] F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In Paliath Narendran and Michael Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Sprinŋger-Verlag.

[4] Adel Bouhoula and Jean-Pierre Jouannaud. Automata-driven automated induction. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 14–25, Warsaw,Poland, June 1997. IEEE Comp. Soc. Press.

[5] S. Boutin. *Réflexions sur les quotients*. thèse d'université, Paris 7, April 1997.

[6] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988.

[7] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Sprinŋger-Verlag, 1990.

[8] H. Geuvers. The church rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Proc. 7th IEEE Symp. Logic in Computer Science, Santa Cruz*, pages 453–460, 1992.

[9] M. Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, Edinburgh university, 1995.

[10] M. Hofmann. A simple model for quotient types. In *proc TLCA'95*, volume 902 of *lncs*. Springer-Verlag, 1995.

[11] B. Jacobs. Quotients in simple type theory. Drafts and Notes, http://www.cwi.nl/ bjacobs/, 1991.

[12] Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, pages 328–345. Sprinŋger-Verlag, 1993. LNCS 664.

[13] G. Malcolm R. Backhouse, P. Chisholm and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1989.

[14] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. Thèse de doctorat, Univ. Paris VII, 1994.

A Example: the integers

We shall now describe by an example the use of normalized types. We define the integers ($\mathbb{Z}$), using a type `Int` that has 3 constructors `O`, `S` and `P`, and a function `nf` that eliminates the useless combinations : `(S (P _))` and `(P (S _))`. To improve readability, we present this example as a pseudo Coq session. Coq is a tool that allows to interactively define, and type, terms of $\mathcal{CIC}$. One way to define terms is to build them with a *proof engine* interactively by means of a set of *tactics*. We will suppose here that it has been extended to $\mathcal{CIC}^{nf}$. Actually, normalized types have only been axiomatized in the real Coq. `Int, nf` and `I` have been defined, in particular `I` has been proved. Of course when we use normalized types, we pretend that normalized types have been implemented in the system Coq, which is not the case (though it will be implemented in a non official version of the system in a short future).

A.1 Basic definitions

We assume that the propositional equality is defined like it is presently in Coq, i.e. as the least reflexive relation:

```
Inductive eq [A:Set; x:A] : A->Prop :=  refl_equal : (eq A x x)
```

We first define the underlying type `Int`:

```
Inductive Int : Set :=  O: Int | S: Int -> Int | P: Int -> Int.
```

Then we define the function `nf`:

```
Fixpoint nf [n:Int]: Int :=
  Cases n of
    O     => O
  | (S a) => Cases a of
               O  => (S O)
             | (S y) =>  Cases (nf a) of O  => (S O)
                                        | (S x) => (S (S x))
                                        | (P x) => x  end
             |(P y) => (nf y) end
  | (P a) =>  ...  end.
```

Notice that it is written for a *head-first* strategy. That way, a term of the form `(nf (P (S (P x))))` will reduce to `(nf (P x))`, which is not the case with the more obvious function (the two terms are only provably equal). This is a weakness of the normalized types that we briefly discuss in the conclusion.

A.2 Definition and use of the normalized type

From now we still present the example in Coq syntax but it is supposed that the system has been modified to deal with normalized types as it is described in this paper. We define the normalized type representing $\mathbb{Z}$:

```
Definition Z: Set := Norm (Int,nf).
```

Assuming that our notion of convertibility has been implemented in the system, `Class(Z,nf,O)` and `Class(Z,nf,(S (P O)))` are convertible, therefore we can make the following proof of $S\ P\ 0 = 0$:

```
Lemma (eq Z Class(Int,nf,O) Class(Int,nf,(S (P O)))).
```

```
Exact (refl_equal Z Class(Int,nf,0)). Save.
```

Let us show now the definition of a function. We first define a function IdInt on the underlying type and then we define IdZ on `Z` using `Elimnorm` and IdInt:

```
Definition IdInt : Int -> Z := [x:Z] Class(Int,nf,x).
Definition IdZ : Z -> Z := [x:Z] Elimnorm(Int,nf,IdInt,x).
```

Now we can apply `IdZ` to several terms and see how it is computed.

```
Eval Compute in (IdZ Class(Int,nf,0))  = Class(Int,nf,0)
Eval Compute in (IdZ Class(Int,nf,(S (P 0))))  = Class(Int,nf,0)
```

Let us follow step by step the second reduction:

`(IdZ Class(Int,nf,(S (P 0))))`

$\rightarrow_\beta$ `Elimnorm(Int,nf,IdInt,Class(Int,nf,(S (P 0))))`

$\rightarrow_{nf}$ `(IdInt (nf (S (P 0))))` $\rightarrow^*$ `(IdInt 0)` $\rightarrow^*$ `Class(Int,nf,0)`

Notice that because of the reduction $\rightarrow_{nf}$, `(S (P 0))` is reduced to `0` before being applied to IdInt, this is why the two terms above are reduced to the same. We see here an example of the way incoherence by ι (explained in section 1) is avoided in our system: *nf is applied before any ι-reduction on a normalized term can occur.*

A.3 Better induction scheme

The following principle is a stronger elimination scheme in case that *nf* is idempotent:

```
Elim_A: (A:Set) (nf: A -> A) ((x:Int) (nf (nf x))= (nf x))
         -> (P: A -> Prop) (H:(s:A) (nf s = s) (P Class(A,nf,s)))
         -> (t:(Norm (A,nf))) (P t)
```

It is easily provable in $\mathcal{CIC}^{nf}$, and we can deduce from it a stronger elimination principle for `Z`. We have to prove is that `nf` is Idempotent, and then instantiate `Elim_A`:

`Lemma I:(x:Int) (nf (nf x))= (nf x).` ...<*tactics*>`Save.`

```
Lemma ElZ:
    (P:Z->Prop)(H:(x:Int)(nf x = x)(P Class(Int,nf,x)))->(t:Z)(P t).
```

which is already a better induction principle. We can build an even more useful one by defining inductive predicate `NormInt` equivalent to the proposition `(nf x = x)`:

```
Inductive pos:Int->Prop:=p0:(pos (S 0)) | pS:(x:Int)(pos x)->(pos (S x))
Inductive neg:Int->Prop:=n0:(neg (P 0)) | pS:(x:Int)(neg x)->(neg (S x))
Inductive NormInt: Int -> Prop :=  norm0  : (NormInt I0)
                                 | normpos: (x:Int) (pos x) (NormInt x)
                                 | normneg: (x:Int) (neg x) (NormInt x).
Lemma Normint_correct : (x:Int) (NormInt x) <-> (nf x = x). ... Save.
```

Finally we can replace one by the other and obtain the well known principle on $\mathbb{Z}$:

```
Lemma IZ:
(P: Z -> Prop) (H:(x:Int) (NormInt x) (P Class(Int,nf,x))) -> (t:Z) (P t)
```

B The calculus of inductive constructions

Here is a short description of the calculus of inductive constructions first defined in [7], following notations of [14] and [12]. We first give the syntax, then the typing and reduction rules.

The Syntax is the following:

Variables : $V ::= x, y, z \dots$

Sorts: $S ::= \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Extern}^5$.

Terms:

$$T ::= V \mid S \mid [V:T]T \mid (V:T)T \mid TT$$
$$\mid \mathsf{Ind}(V:T)(\boldsymbol{T}) \mid \mathsf{Constr}(n,T)\ n \in \mathbb{N} \mid \mathsf{Elim}(T,T,\boldsymbol{T},T)\{\boldsymbol{T}\}$$

Ind, Constr and Elim are respectively the type constructor, term constructor and destructor for inductive types.

Reduction reduction of $\mathcal{CIC}$ is the congruent closure of β and ι-reductions. See [14] for precisions on the term $\Delta[t_1, t_3, t_2]$. It roughly applies the good arguments $(\vec{m})$ to the *branch* (f_k) of the recursive definition.

$$\begin{array}{ll} ([x:t]t_1\ t_2) & \rightarrow_\beta t_1[x\backslash t_2] \\ \mathsf{Elim}(I, Q, \boldsymbol{a}, \mathsf{Constr}(k, I')\boldsymbol{m})\{\boldsymbol{f_i}\} & \rightarrow_\iota (\Delta[C_k(I), f_k, Fun\_Elim(I, Q, \boldsymbol{f_i})]\boldsymbol{m}) \\ & \text{where} \quad I = \mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\} \end{array}$$

The type system

$$(\mathrm{Ax}_1)[] \vdash \mathsf{Set} : \mathsf{Type} \qquad (\mathrm{Ax}_2)[] \vdash \mathsf{Type} : \mathsf{Extern}$$

$$(\text{PROD-S})\frac{\Gamma :: (x:t_1) \vdash t_2 : s}{\Gamma \vdash (x:t_1)t_2 : s} \qquad (\text{LAM-S})\frac{\Gamma \vdash (x:t_1)t_2 : s \qquad \Gamma :: (x:t_1) \vdash t : t_2}{\Gamma \vdash [x:t_1]t : (x:t_1)t_2}$$

$$(\text{W-SET})\frac{\Gamma \vdash t : \mathsf{Set} \quad \Gamma \vdash A : B \quad a \notin \Gamma}{\Gamma :: (a:t) \vdash A : B} \qquad (\text{W-TYPE})\frac{\Gamma \vdash t : \mathsf{Type} \quad \Gamma \vdash A : B \quad \alpha \notin \Gamma}{\Gamma :: (\alpha:t) \vdash A : B}$$

$$(\text{VAR})\frac{\Gamma \vdash t_1 : t_2 \qquad (x:t_1) \in \Gamma}{\Gamma \vdash x : t_1} \qquad (\text{APP})\frac{\Gamma \vdash t_2 : (x:T_1)T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash (t_2\ t_1) : T_2[x \leftarrow t_1]}$$

$$(\text{CONV})\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1 : s \qquad \Gamma \vdash T_2 : s \qquad T_1 \equiv_{\beta\iota} T_2}{\Gamma \vdash t : T_2}$$

$$(\text{IND})\frac{Ar(A, \mathsf{Set}) \qquad \Gamma \vdash A : \mathsf{Type} \qquad \forall i.(\Gamma :: (X:A) \vdash C_i(X) : \mathsf{Set}) \qquad \forall i.constr(C_i(X))}{\Gamma \vdash \mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\} : A}$$

$$(\text{INTRO})\frac{\Gamma \vdash \mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\} : T \qquad 1 \leq n \leq \mid \boldsymbol{C_i(X)} \mid}{(\Gamma \vdash \mathsf{Constr}(n, \mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\}) : C_n(\mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\}))}$$

$$(\text{W-ELIM})\frac{\begin{array}{c} A \equiv (\boldsymbol{x}:\boldsymbol{A})Set \qquad I = \mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\} \qquad \Gamma \vdash \boldsymbol{u} : \boldsymbol{A} \qquad \Gamma \vdash t : (I\ \boldsymbol{u}) \\ \Gamma \vdash Q : (\boldsymbol{x}:\boldsymbol{A})(I\ \boldsymbol{x}) \rightarrow \mathsf{Set} \qquad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i,I)\}) \end{array}}{\Gamma \vdash \mathsf{Elim}(I, Q, \boldsymbol{u}, t)\{\boldsymbol{f_i}\} : (Q\ \boldsymbol{u}\ t)}$$

$$(\text{S-ELIM})\frac{\begin{array}{c} A \equiv (\boldsymbol{x}:\boldsymbol{A})Set \qquad I = \mathsf{Ind}(X:A)\{\boldsymbol{C_i(X)}\} \qquad \Gamma \vdash \boldsymbol{u} : \boldsymbol{A} \qquad \Gamma \vdash t : (I\ \boldsymbol{u}) \\ \Gamma \vdash Q : (\boldsymbol{x}:\boldsymbol{A})(I\ \boldsymbol{x}) \rightarrow \mathsf{Type} \\ \forall i.Small(C_i(X)) \qquad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i,I)\}) \end{array}}{\Gamma \vdash \mathsf{Elim}(I, Q, \boldsymbol{u}, t)\{\boldsymbol{f_i}\} : (Q\ \boldsymbol{u}\ t)}$$

The system is composed of the usual set of typing rules for a pure type system (PTS) and a specific set of rules for inductive types (Ind and below).

The ELIM rules make use of the term $\Delta\{C_i(I), Q, \mathsf{Constr}(i,I)\}$, which is again defined in[14], that builds the type that each branch of a recursive function defined with Elim should have.

The constructions $constr(C_i(X))$, $Ar(A, \mathsf{Set})$ and $Small(C_i(X))$ are very important syntactic conditions that must be satisfied to ensure normalization and coherency of the system. There exact formulation is not very important for us, the important point is that the resulting calculus has nice properties like strong normalization, confluence and subject-reduction, and that normalized types can therefore be define from it.

---

[5] it is possible to replace Extern by a universe hierarchy, actually it is the case for the system implemented in Coq and Lego, it seems not difficult to extend our work to a universe hierarchy.