Coq in Coq

Bruno Barras and Benjamin Werner
INRIA-Rocquencourt

(Received ..... ; Accepted in final form .....)

Abstract. We formalize the definition and the metatheory of the Calculus of Constructions (CC) using the proof assistant Coq. In particular, we prove strong normalization and decidability of type inference. From the latter proof, we extract a certified Objective Caml program which performs type inference in CC and use this code to build a small-scale certified proof-checker.

Key words: Type Theory, proof-checker, Calculus of Constructions, metatheory, strong normalization proof, program extraction.

1. Introduction

1.1. Motivations

This work can be described as the formal certification in Coq of a proof-checker for the Calculus of Constructions (CC). We view it as a first experimental step towards a certified kernel for the whole Coq system, of which CC is a significative fragment. In decidable type theories, a proof-checker is a program which verifies whether a given judgement (input) is valid or not (output). Valid meaning that there exists a derivation for that judgement following the inference rules of the theory. When formulating the specification of that problem in the program extraction paradigm of Coq, it reduces to the (constructive) proof of the decidability for type-checking, which itself is the final consequence of the main metatheoretic results: confluence, subject-reduction and strong normalization.

Because of Gödel's incompleteness theorem, we cannot hope for a strong normalization proof of the whole type theory underlying Coq. Since our primary goal was to explore the feasibility of the software certification process, we chose to leave the normalization result as an axiom for a first try. Once this work had been suitably fulfilled, we decided to go for a formal proof of strong normalization for the Calculus of Constructions, which, to our knowledge, was a première. There are therefore two aspects to the present work:

– A fully certified type-checker for CC.

– A complete formalization of the syntactic metatheory of CC; roughly a formal checking of the results of Coquand's Thesis [5].

1.2. Some preliminary comments

This paper intends to be a kind of informal abstract of a formalized and mechanically checked piece of mathematics; here we try to point out interesting points and difficulties. We might divide the formal objects and results in three families:

- The objects and lemmas appearing in the extracted program, *i.e.* mainly the data types for terms and judgements, some lifting operations on De Bruijn indices and the decidability of type inference.
- The basic metatheoretical results with no computational content (confluence, subject reduction, etc).
- The strong normalization proof.

We believed, and turned out to be right, that Coq was extremely well-suited for, at least, the two first points, since even their informal versions are essentially based on mathematical processes Coq is well-equipped for: inductive definitions and proofs by structural recursion. The strong normalization proof was more of a challenge, since it is generally presented in a more set-theoretical setting, and no thought had yet been given as how to go beyond Altenkirch's formalized normalization proof for system F [1]. Actually, it was indeed necessary to use extensively advanced features of Coq's type theory in order to push our normalization proof through.

In the whole paper, `verbatim` typesetting will be used for Coq-formalized objects. All the definitions, lemmas and theorems are stated together with their respective names in the formal development.

2. Terms of the Calculus of Constructions

In this section we define the syntax, β-reduction and some combinatorial properties on the raw terms. These definitions are straightforward enough to be confident in the fact that they actually implement the intended formalism.

2.1. Syntax of expressions

We consider a λ-calculus with dependent types. As it is now usual, we use the same syntax for terms and types. The correspondence between informal, mathematical and formal notations is described in figure 1.

Before defining the terms, we have to introduce the set of sorts. The sorts are special predefined constants: they are the type of types. For details see [2].

informal	named variables	de Bruijn	formal
Context	Γ	Γ	`e:env`
List item	$\Gamma(n) = (x,t)$	$\Gamma(n) = t$	`(item t e n)`
Item with lift	$\Gamma(n) = (x,t)$	$\uparrow^{n+1}\Gamma(n) = t$	`(item_lift t e n)`
Sorts	Prop, Kind	Prop, Kind	`prop`, `kind`
Variables	x	n if $\Gamma(n) = (x,T)$	`(Ref n)`
Abstraction	$\lambda x:T.M$	$\lambda T.M$	`(Abs T M)`
Application	$(u\ v)$	$(u\ v)$	`(App u v)`
Dependent product	$\Pi x:T.U$	$\Pi T.U$	`(Prod T U)`
Non-dependent product	$A \rightarrow B$	$A \rightarrow B$	`(Prod A (lift (S O) B))`
Lift	T	$\uparrow^n_k T$	`(lift_rec n T k)`
Substitution	$M[x\backslash N]$	$M[k\backslash N]$ if $\Gamma(k) = (x,T)$	`(subst_rec N M k)`

Figure 1. Correspondence between various notations

DEFINITION 1 (type `sort`). The set of sorts of the Calculus of Constructions has two elements: Kind and Prop.

$$\textsc{Sort} := \text{Kind} \mid \text{Prop}$$

DEFINITION 2 (type `term`). The syntactic class $\textsc{Term}$ is defined by the following grammar:

$$\textsc{Term} := s \mid n \mid \lambda T_1.T_2 \mid (T_1\ T_2) \mid \Pi T_1.T_2$$

where $T_1, T_2 \in \textsc{Term}$, $s \in \textsc{Sort}$ and $n \in \mathbb{N}$.

Let us simply recall that $\Pi T_1.T_2$ denotes the dependent function type from T_1 to T_2.

Since we use de Bruijn notation for bindings, no variable names are needed in the context. The latter are simply term lists.

DEFINITION 3 (type `env`). Contexts are term lists.

We note $|\Gamma|$ the length of the context Γ. To denote the n-th element of a context Γ (the rightmost element has rank 0), we use the notation $\Gamma(n)$, which implicitly assumes that $n < |\Gamma|$.

2.2. Reduction rules

In this calculus, we only consider the β-reduction. To define this reduction rule, we have to define substitution first. de Bruijn notation requires

Lift	Substitution
$\uparrow^n_k s \;=\; s$	$s[k\backslash N] \;=\; s$
$\uparrow^n_k i \;=\; \begin{cases} i+n & \text{if } i \geq k \\ i & \text{if } i < p \end{cases}$	$i[k\backslash N] \;=\; \begin{cases} i-1 & \text{if } i > k \\ \uparrow^k N & \text{if } i = k \\ i & \text{if } i < k \end{cases}$
$\uparrow^n_k \lambda T.t \;=\; \lambda \uparrow^n_k T. \uparrow^n_{k+1} t$	$(\lambda T.t)[k\backslash N] \;=\; \lambda T[k\backslash N].t[k+1\backslash N]$
$\uparrow^n_k (u\ v) \;=\; (\uparrow^n_k u\ \uparrow^n_k v)$	$(u\ v)[k\backslash N] \;=\; (u[k\backslash N]\ v[k\backslash N])$
$\uparrow^n_k \Pi T.U \;=\; \Pi \uparrow^n_k T. \uparrow^n_{k+1} U$	$(\Pi T.U)[k\backslash N] \;=\; \Pi T[k\backslash N].U[k+1\backslash N]$

Figure 2. Definition of lift and substitution

defining another function on terms: the relocation of de Bruijn indices, also called "lifting".

2.2.1. *Lift, substitution and β-reduction*

DEFINITION 4 (`lift_rec`, `lift`, `subst_rec`, `subst`). Given a term M and integers n and k, we define $\uparrow^n_k M$ `(lift_rec n M k)` as the term M where all the indexes greater that k are lifted by n. We write $\uparrow^n M$ for the lift of all variables (a shortcut for $\uparrow^n_0 M$).

$M[k\backslash N]$ (written `(subst_rec N M k)` in Coq) stands for the substitution of the variable k by the term N in M. The precise definitions are given in figure 2.

So far, we defined the objects the program we want to verify will deal with. From here on up to section 7, we will introduce notions and results without any computational content, which means that they will not appear in the extracted program.

DEFINITION 5 (predicate `red1`). The one-step β-reduction, written $\triangleright_\beta$, is the smallest binary relation on terms closed by the inference rules on figure 3.

DEFINITION 6 (predicates `red`, `conv`). β-reduction of arbitrary many steps will be written $\triangleright^*_\beta$. The notation for the β-conversion will be $\approx_\beta$.

In a general way, for any relation R, we write R^* its reflexive transitive closure and R^+ its transitive closure.

2.2.2. *Algebraic properties of* `lift` *and* `subst`

The following are basic properties of the lifting and substitution operations. They have already been proved by Huet [13] for the pure λ-calculus and the proofs translate easily to annotated terms.

$$\text{(Beta)}\ \frac{}{(\lambda T.M\ N) \rhd_\beta M[0\backslash N]}$$

$$\text{(Abs-L)}\ \frac{M \rhd_\beta M'}{\lambda M.N \rhd_\beta \lambda M'.N} \qquad \text{(Abs-R)}\ \frac{M \rhd_\beta M'}{\lambda N.M \rhd_\beta \lambda N.M'}$$

$$\text{(App-L)}\ \frac{M_1 \rhd_\beta N_1}{(M_1\ M_2) \rhd_\beta (N_1\ M_2)} \qquad \text{(App-R)}\ \frac{M_2 \rhd_\beta N_2}{(M_1\ M_2) \rhd_\beta (M_1\ N_2)}$$

$$\text{(Prod-L)}\ \frac{M_1 \rhd_\beta N_1}{\Pi M_1.M_2 \rhd_\beta \Pi N_1.M_2} \qquad \text{(Prod-R)}\ \frac{M_2 \rhd_\beta N_2}{\Pi M_1.M_2 \rhd_\beta \Pi M_1.N_2}$$

Figure 3. The β-reduction relation

LEMMA 7.

`lift_rec0` :	$\uparrow^0_k M = M$
`simpl_lift_rec` :	$\uparrow^p_i (\uparrow^n_k M) = \uparrow^{p+n}_k M$ if $k \le i \le k+n$
`permute_lift_rec` :	$\uparrow^p_i (\uparrow^n_k M) = \uparrow^n_{p+k} (\uparrow^p_i M)$ if $i \le k$
`simpl_subst_rec` :	$(\uparrow^{n+1}_k M)[p\backslash N] = \uparrow^n_k M$ if $k \le p \le n+k$
`commut_lift_subst_rec` :	$\uparrow^n_k (M[p\backslash N]) = (\uparrow^n_k M)[n+p\backslash N]$ if $k \le p$
`distr_lift_subst_rec` :	$\uparrow^n_{p+k} (M[p\backslash N]) = (\uparrow^n_{p+k+1} M)[p\backslash \uparrow^n_k N]$
`distr_subst_rec` :	$(M[p\backslash N])[p+n\backslash P] = (M[p+n+1\backslash P])[p\backslash N[n\backslash P]]$

Remark 8. The last lemma seems easier to read with named variables notations:

$$M[x\backslash N][y\backslash P] = M[y\backslash P][x\backslash N[y\backslash P]] \text{ if } x \neq y\ \wedge\ x \text{ not free in } P$$

But this apparent simplicity is misleading; the side condition on variable names make it much more difficult to use than expected.

2.3. Strongly normalizing terms

A term is *strongly normalizing* if and only if there is no infinite reduction path starting from it. The following definition is well-known since [14] and expresses that, for a relation R, there is no infinite decreasing sequence starting from t:

DEFINITION 9 (predicate `Acc`) . The set Acc_R is the smallest set verifying:

$$\forall t.\ (\forall u.\ u\ R\ t \Rightarrow u \in \mathrm{Acc}_R) \Rightarrow t \in \mathrm{Acc}_R.$$

$$\begin{tikzcd}
 & x & \\
y \arrow[ur, "R1"] & & y' \arrow[ul, dashed, "R2"'] \\
 & z \arrow[ul, "R2"] \arrow[ur, dashed, "R1"'] &
\end{tikzcd}$$

Figure 4. Predicate `commut R1 R2` $\equiv (R2; R1 \subseteq R1; R2)$

DEFINITION 10 (predicate `sn`). The set $\mathcal{SN}$ of strongly normalizing terms is defined as $\mathrm{Acc}_{\rhd_\beta^{-1}}$.

Note that this is actually equivalent to Altenkirch's inductive formulation [1].

The following definition of normal terms seems to be the simplest and emphasizes that they are a particular case of strongly normalizing terms.

DEFINITION 11 (predicate `normal`). A term t is said to be normal if and only if it admits no reduct:

$$\forall u.\ t \rhd_\beta u \Rightarrow \bot.$$

The property for two relations to commute will mainly be used in the confluence proof, but is already useful here:

$$\forall (x, y, z).\ z\ R_2\ y\ \wedge\ y\ R_1\ x \Rightarrow \exists y'.\ (z\ R_1\ y'\ \wedge\ y'\ R_2\ x)$$

The direct subterm relation $\subset_{st}$ is defined straightforwardly. It can be postponed with respect to β-reduction:

LEMMA 12 (`commut_red1_subterm`). *The relations $\subset_{st}$ and the symmetric of β-reduction commute.*

Which allows to prove:

LEMMA 13 (`subterm_sn`). *The subterm B of a strongly normalizing term A is strongly normalizing:*

$$A \in \mathcal{SN}\ \wedge\ B \subset_{st} A \Rightarrow B \in \mathcal{SN}$$

$$(\textsc{Beta'})\ \frac{M \rhd_{/\!/} M' \qquad N \rhd_{/\!/} N'}{(\lambda T.M\ N) \rhd_{/\!/} M'[0\backslash N']}$$

$$(\textsc{Srt})\ \frac{}{s \rhd_{/\!/} s} \qquad (\textsc{Ref})\ \frac{}{n \rhd_{/\!/} n}$$

$$(\textsc{Abs})\ \frac{M \rhd_{/\!/} M' \qquad T \rhd_{/\!/} T'}{\lambda T.M \rhd_{/\!/} \lambda T'.M'} \qquad (\textsc{App})\ \frac{M \rhd_{/\!/} M' \qquad N \rhd_{/\!/} N'}{(M\ N) \rhd_{/\!/} (M'\ N')}$$

$$(\textsc{Prod})\ \frac{M \rhd_{/\!/} M' \qquad N \rhd_{/\!/} N'}{\Pi M.N \rhd_{/\!/} \Pi M'.N'}$$

Figure 5. The parallel β-reduction relation

2.4. Church-Rosser and Confluence

It is essential that β-reduction verifies the Church-Rosser property, since it is necessary to several key-results: type uniqueness, subject-reduction and decidability of typing. We use a very traditional technique, first proving confluence by the Tait–Martin-Löf method. One defines *parallel β-reduction* which is a strongly confluent relation whose reflexive transitive closure is equivalent to β-reduction.

The formalization is surprinsingly close to the usual informal proof.

DEFINITION 14 (predicate `par_red1`). The parallel β-reduction, written $\rhd_{/\!/}$, is defined as the smallest relation on terms closed by the rules in figure 5.

We define the strong confluence property, expressed in terms of commutation, but it expands to the usual definition.

DEFINITION 15 (predicate `str_confluent`). A relation R is strongly confluent if and only if R commutes with its symmetric:

$$R^{-1}; R \subseteq R; R^{-1}.$$

LEMMA 16 (`str_confluence_par_red1`) . *The parallel β-reduction is strongly confluent.*

LEMMA 17 (`confluence_red`) . *The β-reduction relation is confluent, i.e. its reflexive transitive closure is strongly confluent.*

As we said in the beginning of this section, we can prove that the Church-Rosser property holds, by an easy induction.

$$(\text{WF-[]})\ \frac{}{[\,]\vdash} \qquad (\text{WF-VAR})\ \frac{\Gamma\vdash T:s}{\Gamma;T\vdash}(s\in\textsc{Sort})$$

$$(\textsc{Prop})\ \frac{\Gamma\vdash}{\Gamma\vdash\mathrm{Prop}:\mathrm{Kind}} \qquad (\textsc{Var})\ \frac{\Gamma\vdash \qquad \uparrow^{n+1}\Gamma(n)=T}{\Gamma\vdash n:T}$$

$$(\textsc{Abs})\ \frac{\Gamma\vdash T:s_1 \qquad \Gamma;T\vdash M:U \qquad \Gamma;T\vdash U:s_2}{\Gamma\vdash\lambda T.M:\Pi T.U}(s_1,s_2\in\textsc{Sort})$$

$$(\textsc{App})\ \frac{\Gamma\vdash v:V \qquad \Gamma\vdash u:\Pi V.T}{\Gamma\vdash(u\ v):T[0\backslash v]}$$

$$(\textsc{Prod})\ \frac{\Gamma\vdash T:s_1 \qquad \Gamma;T\vdash U:s_2}{\Gamma\vdash(\Pi T.U):s_2}\ (s_1,s_2\in\textsc{Sort})$$

$$(\textsc{Conv})\ \frac{\Gamma\vdash M:U \qquad \Gamma\vdash V:s \qquad U\approx_\beta V}{\Gamma\vdash M:V}\ (s\in\textsc{Sort})$$

Figure 6. Typing Rules for the Calculus of Constructions

THEOREM 18 (`church_rosser`). *The β-reduction satisfies the Church-Rosser property:*

$$\forall u,v\in\textsc{Term}.\ u\approx_\beta v\Rightarrow\exists t\in\textsc{Term}.\ (u\rhd_\beta^* t\ \wedge\ v\rhd_\beta^* t).$$

Remark 19. We proved a non computational version of Church-Rosser. A computational proof would be much more difficult, because it precludes reasoning by induction on the hypothesis $u\approx_\beta v$, which is not computational. Such a proof would give an algorithm computing a common reduct of two convertible terms. This will be made possible in section 7.1.

COROLLARY 20 (`inv_conv_prod_{l, r}`). *The uniqueness of product formation property holds, i.e. if two products are convertible, their left (resp. right) subterms are convertible:*

$$\Pi A.C\approx_\beta\Pi B.D\Rightarrow\begin{cases}A\approx_\beta B\\ C\approx_\beta D\end{cases}$$

3. The Rules

3.1. Definition

As usual, derivability is defined as an inductive predicate; each inference rule being read as a clause. We chose to define two kinds of judgements by mutual induction:

– $\Gamma \vdash$ to express that the context Γ is well-formed

– $\Gamma \vdash t : T$ to express that the term t is of type T in Γ.

DEFINITION 21 (predicates `wf`, `typ`). We define the two sets of derivable judgements as the smallest sets respectively closed by the inference rules of figure 6.

Note that it would not be necessary to require that T is well-typed in the formation rules of the product and the λ-abstraction. This way we stick to the usual PTS formulation.

Loosening the conversion rule in a similar way (requiring $U \triangleright_\beta^* V$ instead of U convertible with a well-formed type V) is more problematic with respect to subject reduction; see [18].

3.2. Inversion lemmas

The context of any derivable judgement is well-formed. In other words:

LEMMA 22 (`typ_wf`). *The rule*

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash}$$

is admissible.

LEMMA 23 (`wf_sort`). *The terms of a well-formed environment are well-typed:*

$$\Gamma; T; \Delta \vdash \Rightarrow \exists s \in \textsc{Sort}.\ \Gamma \vdash T : s$$

Both proofs go easily by induction over the structure of the derivation.

The inversion lemmas state that the derivation and the typed term have the same shape. One consequence is that every subterm of a well-typed term is well-typed. But the main point is that we can use recursion on a term instead of an induction on a logical hypothesis. That way, we are able to perform computational proofs.

THEOREM 24 (`inv_typ_`{`kind`, `prop`, `ref`, `abs`, `app`, `prod`}).

$$\begin{aligned}
\Gamma \vdash \mathrm{Kind} : T &\Rightarrow \bot \\
\Gamma \vdash \mathrm{Prop} : T &\Rightarrow T \approx_\beta \mathrm{Kind} \\
\Gamma \vdash n : T &\Rightarrow T \approx_\beta \uparrow^{n+1}\Gamma(n) \\
\Gamma \vdash \lambda A.M : U &\Rightarrow \exists T \in \textsc{Term}.\ \exists s_1, s_2 \in \textsc{Sort}.\ \begin{cases} \Gamma \vdash A : s_1 \\ \Gamma; A \vdash M : T \\ \Gamma; A \vdash T : s_2 \\ U \approx_\beta \Pi A.T \end{cases}
\end{aligned}$$

$$\Gamma \vdash (u\ v) : T \ \Rightarrow\ \exists V, U_r \in \textsc{Term}.\ \begin{cases} \Gamma \vdash u : \Pi V.U_r \\ \Gamma \vdash v : V \\ T \approx_\beta U_r[0\backslash v] \end{cases}$$

$$\Gamma \vdash \Pi u.v : T \ \Rightarrow\ \exists s_1, s_2 \in \textsc{Sort}.\ \begin{cases} \Gamma \vdash T : s_1 \\ \Gamma; T \vdash U : s_2 \\ T \approx_\beta s_2 \end{cases}$$

COROLLARY 25 (`inv_typ_conv_kind`) . *No term convertible with* Kind *is well-typed*

$$t \approx_\beta \text{Kind} \Rightarrow \forall T.\ \neg(\Gamma \vdash t : T).$$

4. Basic Metatheory

This section is devoted to the usual elementary properties of the type system. All these results are of combinatorial nature and are the de Bruijn counterparts of what can be found in, say, [9, 2] and others. In some cases, the order of the lemmas might be changed, due to slight differences in the formulation of the typing rules.

4.1. Thinning lemma

This lemma (often also called "weakening") simply states that typing for a term is preserved if new assumptions (i.e. variables) are added to the environment. The exact formulation is slightly more complex in its de Bruijn version, since inserting new assumptions induces some lifting operations.

DEFINITION 26 (predicate `ins_in_env`). We first extend the definition of lifting to contexts by the two inductive clauses:

$$\begin{aligned} [\,]^+ &= [\,] \\ (\Delta; T)^+ &= \Delta^+; \uparrow^1_{|\Delta|} T \end{aligned}$$

The predicate stating that $(\Gamma; T; \Delta^+)$ is obtained by inserting T in $(\Gamma; \Delta)$ is then defined as a Prolog-style inductive predicate by the clause:

$$\frac{}{(\texttt{ins\_in\_env}\ T\ |\Delta|\ (\Gamma; \Delta)\ (\Gamma; T; \Delta^+))}$$

LEMMA 27 (`thinning`) . *The following rule is admissible:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma; A \vdash}{\Gamma; A \vdash \uparrow^1 t : \uparrow^1 T}$$

Proof In two steps; one first shows a stronger version, not assuming anymore that A is the last assumption of the context (induction loading):

$$\frac{\Gamma;\Delta\vdash t:T \qquad \Gamma;A;\Delta^{+}\vdash}{\Gamma;A;\Delta^{+}\vdash\uparrow^{1}_{|\Delta|}t:\uparrow^{1}_{|\Delta|}T}$$

This is done by induction over the first hypothesis $\Gamma\vdash t:T$. The case of application is treated using the algebraic property `distr_lift_subst`.

The result follows by taking $\Delta=[\,]$. ■

4.2. Substitution lemma

Typing is preserved by (well-typed) substitution. Again, we first have to extend the definition of substitution to contexts, taking care of reallocation of de Bruijn indexes.

DEFINITION 28 (`sub_in_env`). Informally, substitution in an environment is described by the two following rules:

$$\begin{aligned}[\,][t] &= [\,]\\ (\Delta;T)[t] &= \Delta[t];T[|\Delta|\backslash t]\end{aligned}$$

The formal definition is given by the following clause:

$$\frac{}{(\texttt{sub\_in\_env}\ M\ T\ |\Delta|\ (\Gamma;T;\Delta)\ (\Gamma;\Delta[M]))}$$

THEOREM 29 (`substitution`). *The rule:*

$$\frac{\Gamma;T\vdash u:U \qquad \Gamma\vdash d:T}{\Gamma\vdash u[0\backslash d]:U[0\backslash d]}$$

is admissible.

Proof Similar to the proof of the thinning lemma. The first thing to do is show that:

$$\frac{\Gamma;T;\Delta\vdash u:U \qquad \Gamma\vdash d:T \qquad \Gamma;(\Delta[d])\vdash}{\Gamma;(\Delta[d])\vdash u[|\Delta|\backslash d]:U[|\Delta|\backslash d]}$$

Here again, we apply this result with $\Delta=[\,]$. ■

4.3. Type uniqueness results

An easy consequence of the inversion lemmas is that a term has at most one type, up to β-conversion:

THEOREM 30 (`typ_unique`).

$$\left.\begin{array}{l}\Gamma \vdash t : T \\ \Gamma \vdash t : U\end{array}\right\} \Rightarrow T \approx_\beta U$$

THEOREM 31 (`type_case`). *Every inhabited type is either* Kind *or a well-formed term itself:*

$$\Gamma \vdash t : T \Rightarrow (\exists s \in \textsc{Sort}.\ \Gamma \vdash T : s) \vee (T = \text{Kind})$$

PROOF

By induction on the derivation of $\Gamma \vdash t : T$. The only non-trivial case is that of application in which case one applies the induction hypothesis to the left subterm: either its type is of type a sort and the substitution lemma allows to conclude, or its type is Kind, which is absurd since it is not convertible to a product. ■

4.4. Subject reduction

For induction loading it is necessary to also state subject reduction for the context.

DEFINITION 32 (`red1_in_env`). We define β-reduction as the smallest relation over context verifying:

$$\frac{t \rhd_\beta u}{\Gamma; t \rhd_\beta \Gamma; u} \qquad \frac{\Gamma \rhd_\beta \Gamma'}{\Gamma; t \rhd_\beta \Gamma'; t}$$

THEOREM 33 (`subject_reduction`). *The rule*

$$\frac{\Gamma \vdash t : T \qquad t \rhd_\beta^* u}{\Gamma \vdash u : T}$$

is admissible.

PROOF

We first prove: $\dfrac{\Gamma \vdash t : T \qquad \Gamma \rhd_\beta \Gamma' \qquad \Gamma' \vdash}{\Gamma' \vdash t : T}$. ■

COROLLARY 34 (`typ_conv_conv`). *Two well-typed β-convertible terms have the same types:*

$$\left.\begin{array}{r}u \approx_\beta v \\ \Gamma \vdash u : U \\ \Gamma \vdash v : V\end{array}\right\} \Rightarrow U \approx_\beta V$$

5. Type approximations

This section prove results needed in the normalization proof. We introduce a classification of terms which is an approximation of its type. After defining a function that computes this approximation, we prove stability results, and then soundness with respect to typing.

5.1. Definition of classes

The `type_case` lemma states that any inhabited type is either typed by a sort, or Kind itself. A well-known consequence is that one can distinguish 3 levels of terms:

– those of type Kind: they make up the level of *kinds*,

– those whose type has type Kind: they form the level of *predicates*,

– those whose type has type Prop: they constitute the level of *terms*.

This corresponds to the alternative way of defining the terms of the Calculus of Constructions, introducing 3 syntactic classes:

$$\begin{array}{rcl} K & := & \mathrm{Prop} \mid \Pi T.K \mid \Pi K_1.K_2 \\ T & := & i \in \mathbb{N} \mid \Pi T_1.T_2 \mid \Pi K.T \mid \lambda T_1.T_2 \mid (T\ t) \mid \lambda K.T \mid (T_1\ T_2) \\ t & := & i \in \mathbb{N} \mid \lambda T.t \mid (t_1\ t_2) \mid \lambda K.t \mid (t\ T) \end{array}$$

We can see that kinds are functions types, and if we forget dependent types (i.e. erase the leaf T in nodes of the form $\Pi T.K$), kinds are simply binary trees, called *kind skeleton*. This kind skeleton is a representation of the arity of this kind, which is the significative part of kinds and predicates.

DEFINITION 35 (type `skel`). Kind skeletons are isomorphic to binary trees:

$$\textsc{Skel} := \star \mid (S_1 \rightarrow S_2)$$

DEFINITION 36 (types `class`, `cls`). The set of classes has 3 levels: kind, predicate and term; the first two being annotated by a skeleton:

$$\textsc{Class} := (\mathrm{Knd}\ S) \mid (\mathrm{Typ}\ S) \mid \mathrm{Trm}$$

where S is a skeleton.

We also consider lists of classes, the counterpart of contexts. They will generally be noted Σ.

The kind skeleton annotating the predicate classes is intended to be the skeleton of the type of this predicate (which is a kind).

DEFINITION 37 (`cv_skel`, `typ_skel`). We define projections from classes to skeletons:

$$\mathrm{Knd}^{-1}(c) = \begin{cases} S \text{ if } c = (\mathrm{Knd}\ S) \\ \bigstar \text{ otherwise} \end{cases}$$

$$\mathrm{Typ}^{-1}(c) = \begin{cases} S \text{ if } c = (\mathrm{Typ}\ S) \\ \bigstar \text{ otherwise} \end{cases}$$

DEFINITION 38 (`cl_term`, `class_env`). Given the classes of free variables (Σ), the class of a term is computed according to the following rules:

$$\begin{aligned}
\mathrm{Cl}_\Sigma(s) &= (\mathrm{Knd}\ \bigstar) \\
\mathrm{Cl}_\Sigma(n) &= \begin{cases} (\mathrm{Typ}\ S) \text{ if } \Sigma(n) = (\mathrm{Knd}\ S) \\ Trm \text{ otherwise} \end{cases} \\
\mathrm{Cl}_\Sigma(\lambda A.M) &= \text{match } (\mathrm{Cl}_\Sigma(A), \mathrm{Cl}_{\Sigma;\mathrm{Cl}_\Sigma(A)}(M)) \text{ with} \\
&\quad \mid (\_, (\mathrm{Knd}\ \_)) \mapsto (\mathrm{Knd}\ \bigstar) \\
&\quad \mid ((\mathrm{Knd}\ S_1), (\mathrm{Typ}\ S_2)) \mapsto (\mathrm{Typ}\ (S_1 \to S_2)) \\
&\quad \mid (\_, c) \mapsto c \\
\mathrm{Cl}_\Sigma((u\ v)) &= \text{match } (\mathrm{Cl}_\Sigma(u), \mathrm{Cl}_\Sigma(v)) \text{ with} \\
&\quad \mid ((\mathrm{Knd}\ \_), \_) \mapsto (\mathrm{Knd}\ \bigstar) \\
&\quad \mid ((\mathrm{Typ}\ (S_1 \to S_2)), (\mathrm{Typ}\ \_)) \mapsto (\mathrm{Typ}\ S_2) \\
&\quad \mid (c, \_) \mapsto c \\
\mathrm{Cl}_\Sigma(\Pi T.U) &= \text{match } (\mathrm{Cl}_\Sigma(T), \mathrm{Cl}_{\Sigma;\mathrm{Cl}_\Sigma(T)}(U)) \text{ with} \\
&\quad \mid ((\mathrm{Knd}\ S_1), (\mathrm{Knd}\ S_2)) \mapsto (\mathrm{Knd}\ (S_1 \to S_2)) \\
&\quad \mid (\_, c) \mapsto c
\end{aligned}$$

This definition extends straightforwardly to contexts:

$$\begin{aligned}
\mathrm{Cl}([\,]) &= [\,] \\
\mathrm{Cl}(\Gamma; T) &= (\mathrm{Cl}(\Gamma); \mathrm{Cl}_\Gamma(T))
\end{aligned}$$

We note $\mathrm{Cl}_\Gamma(T)$ for $\mathrm{Cl}_{\mathrm{Cl}(\Gamma)}(T)$.

5.2. Loose stability results

We prove stability of class by lifting, substitution, reduction, and soundness w.r.t. typing. In this section, these results are stated for level and kind skeleton; they are necessary to prove similar lemmas for the predicate skeleton in section 5.3.

We define the *loose equality* on classes that consists in neglecting the predicate skeletons:

DEFINITION 39 (predicate `loose_eqc`). The loose equality is defined as follows:

$$(\text{Knd } S) \simeq (\text{Knd } S) \qquad (\text{Typ } S_1) \simeq (\text{Typ } S_2) \qquad \text{Trm} \simeq \text{Trm}$$

LEMMA 40 (`loose_eqc_stable`).

$$\Sigma \simeq \Sigma' \Rightarrow \text{Cl}_\Sigma(T) \simeq \text{Cl}_{\Sigma'}(T)$$

DEFINITION 41 (predicate `adj_cls`). The loose order on classes only considers class level:

$$(\text{Typ } S_1) \sqsubset (\text{Knd } S_2) \qquad \text{Trm} \sqsubset (\text{Typ } S)$$

LEMMA 42 (`cl_term_subst`). *The class is stable by substitution:*

$$\text{Cl}_\Sigma(N) \sqsubset c \Rightarrow \text{Cl}_{\Sigma;c;\Sigma'}(M) \simeq \text{Cl}_{\Sigma;\Sigma'}(M[|\Sigma'|\backslash N])$$

LEMMA 43 (`class_`{`knd`, `typ`, `trm`}). *For every derivable judgement* $\Gamma \vdash M : T$,

$$\begin{aligned} T = \text{Kind} \ &\Rightarrow\ \text{Cl}_\Gamma(M) = (\text{Knd Knd}^{-1}(\text{Cl}_\Gamma(M))) \\ \Gamma \vdash T : \text{Kind} \ &\Rightarrow\ \text{Cl}_\Gamma(M) = (\text{Typ Typ}^{-1}(\text{Cl}_\Gamma(M))) \\ \Gamma \vdash T : \text{Prop} \ &\Rightarrow\ \text{Cl}_\Gamma(M) = \text{Trm} \end{aligned}$$

Each lemma uses the previous one.

COROLLARY 44 (`cl_term_sound`). *The loose class order is sound w.r.t. the typing rules:*

$$\Gamma \vdash t : T \ \wedge\ \Gamma \vdash T : K \Rightarrow \text{Cl}_\Gamma(t) \sqsubset \text{Cl}_\Gamma(T)$$

5.3. Strict stability results

We now consider a more precise order on classes, taking into account the following facts:

– the skeletons of a predicate and his type are the same,

– the types of elements of level Trm are predicates of skeleton $\bigstar$, since their type is Prop.

DEFINITION 45 (predicate `typ_cls`). The strict class order:

$$(\text{Typ } S) : (\text{Knd } S) \qquad \text{Trm} : (\text{Typ } \bigstar)$$

LEMMA 46 (`class_subst`) . *Stability of class by substitution.*

$$\mathrm{Cl}_{\Sigma}(N) : c \Rightarrow \mathrm{Cl}_{\Sigma;c;\Sigma'}(M) = \mathrm{Cl}_{\Sigma;\Sigma'}(M[|\Sigma'|\backslash N])$$

THEOREM 47 (`class_sound`) . *The strict class order is sound w.r.t. the typing rules:*

$$\Gamma \vdash M : T \ \wedge\ \Gamma \vdash T : K \Rightarrow \mathrm{Cl}_{\Gamma}(M) : \mathrm{Cl}_{\Gamma}(T)$$

6. Strong normalization proof

The normalization proof certainly is the part of the work which was the most difficult to adapt to type-theory. Let us recall the three essential steps of a reducibility proof:

– for any type T, one defines a set of terms $[\![T]\!]$ interpreting it.

– one verifies that $[\![T]\!] \subset \mathcal{SN}$.

– Normalization then follows from the soundness of the interpretation: $t : T \Rightarrow t \in [\![T]\!]$.

Altenkirch [1] has shown the normalization property for system F in Lego. From there, encoding a proof for the Calculus of Constructions presented two difficulties.

The first is to deal with dependent types, i.e. many redexes may actually occur inside types. This was not too painful to take care of: it is well-known that, in Calculus of Constructions, one might forget the dependency of types w.r.t. terms and obtain well-formed judgements in F_ω. A consequence is that $[\![T]\!]$ might not depend of the terms; from this point of view, our work is similar to [8].

The second problem is the possibility to define, for instance, functions from types to types. Such objects have to be interpreted by mappings associating sets of terms to sets of terms. In other words, the type of the interpretation of T depends, in a very strong way, of the class of T. This is the main reason for the next paragraph.

In some cases, proofs were also a little more tedious that in set theory, since the intuitive proof makes intensive use of partial functions.

6.1. Candidates

We introduce the type of the interpretations of predicates.

6.1.1. *Reducibility schemes*

DEFINITION 48 (type `Can`). We define a function associating a type to every skeleton by structural recursion:

$$\mathcal{C}_\star \overset{\text{def}}{\equiv} \mathcal{P}(\textsc{Term})$$
$$\mathcal{C}_{(S_1 \to S_2)} \overset{\text{def}}{\equiv} \mathcal{C}_{S1} \to \mathcal{C}_{S_2}$$

($\mathcal{P}$ denotes the powerset operator)

We consider an extensional equality on these schemes:

DEFINITION 49 (predicates `eq_cand`, `eq_can`). For any skeleton S we define a binary predicate $\overset{S}{=}$ over $\mathcal{C}_S$ by structural recursion:

$$C_1 \overset{\star}{=} C_2 \overset{\text{def}}{\equiv} \forall t.\ t \in C_1 \Leftrightarrow t \in C_2$$
$$C_1 \overset{(S_1 \to S_2)}{=} C_2 \overset{\text{def}}{\equiv} \forall X_1, X_2.\ X_1 \overset{S_1}{=} X_1 \wedge X_2 \overset{S_1}{=} X_2 \wedge X_1 \overset{S_1}{=} X_2 \Rightarrow C_1(X_1) \overset{S_2}{=} C_2(X_2)$$

This "equality" is only a partial equivalence relation and is not reflexive (next lemma). The schemes belonging to the domain of $\overset{S}{=}$ (i.e. schemes C such that $C \overset{S}{=} C$) will be called *invariant*. In what follows we will be only interested by invariant schemes.

LEMMA 50 (`eq_can_sym`, `eq_can_trans`). *Extentional equality is transitive and symmetric over the sets of invariant schemes.*

6.1.2. *Higher order reducibility candidates*

We can now define the notion of reducibility candidate, generalizing it to higher-order schemes. On $\mathcal{C}_\star$, we use Girard's original definition of candidates [11] like in [1]; alternative definitions like saturated sets would probably also work.

Following Girard, a term is said to be *neutral* if it is not an abstraction. We write this set $\mathcal{N}$.

DEFINITION 51 (predicates `is_cand`, `is_can`). For every skeleton S we define the set $\mathcal{CR}_S$ of candidates of order S by structural recursion:

- a scheme X of order $\bigstar$ (i.e. a set of terms) is a candidate, if and only if it verifies the three following closure conditions:

$$\begin{array}{l} X \subset \mathcal{SN} \\ t \in X \ \wedge \ t \rhd_\beta u \Rightarrow u \in X \\ t \in \mathcal{N} \ \wedge \ (\forall u.\ t \rhd_\beta u \Rightarrow u \in X) \Rightarrow t \in X \end{array}$$

- a scheme of order $(S_1 \rightarrow S_2)$ is a candidate if and only if it maps invariant candidates (of order S_1) to candidates (of order S_2).

We write $\mathcal{ICR}_S$ for the set of invariant elements of $\mathcal{CR}_S$.

The elements of $\mathcal{CR}_\bigstar$ are Girard's reducibility candidates. The following results are usual and easy:

LEMMA 52 (`var_in_cand`,`clos_red_star`, `cand_sat`). *For any $C \in \mathcal{CR}_\bigstar$, any variable n and terms t, t', u, v:*

$$\begin{array}{l} n \in C \\ t \in C \wedge t \rhd_\beta^* t' \Rightarrow t' \in C \\ t[0\backslash u] \in C \wedge u, v \in \mathcal{SN} \Rightarrow (\lambda v.t\ u) \in C \end{array}$$

6.1.3. *Canonical candidates*
For every skeleton we define a canonical candidate.

DEFINITION 53 (`default_can`) . By recursion:

$$\mathrm{C}^{\mathrm{d}}_\bigstar \stackrel{\mathrm{def}}{\equiv} \mathcal{SN}$$
$$\mathrm{C}^{\mathrm{d}}_{(S_1 \rightarrow S_2)} \stackrel{\mathrm{def}}{\equiv} \lambda_{\mathcal{C}_{S_1}} X.\ \mathrm{C}^{\mathrm{d}}_{S_2}$$

LEMMA 54 (`def_inv`, `def_can_cr`). *The canonical candidate is actually an invariant higher order candidate:*

$$\mathrm{C}^{\mathrm{d}}_S \in \mathcal{ICR}_S$$

6.1.4. *Product of candidates*
A function type $A \rightarrow B$ is generally interpreted as the set of terms mapping elements of the interpretation of A to the interpretation of B. The following generalizes this to higher-order schemes.

DEFINITION 55 (`Pi`). Let C be a scheme of order $\bigstar$ and F a scheme of order $(S \to \bigstar)$. Their product is a scheme of order $\bigstar$ defined by:

$$\Pi_S(C,F) = \{t \in \textsc{Term} \mid \forall u \in C.\ \forall_{\mathcal{ICR}_S} X.\ (t\ u) \in F(X)\}$$

LEMMA 56 (`eq_can_Pi`, `is_can_Pi`). *The product formation preserves extentional equality and the notion of higher-order candidate.*

$$\begin{array}{c} C_1 \overset{\bigstar}{=} C_2 \ \wedge\ F_1 \overset{(S\to\bigstar)}{=} F_2 \ \Rightarrow\ \Pi_S(C_1,F_1) \overset{\bigstar}{=} \Pi_S(C_2,F_2) \\ C \in \mathcal{CR}_\bigstar \ \wedge\ F \in \mathcal{CR}_{(S\to\bigstar)} \ \Rightarrow\ \Pi_S(C,F) \in \mathcal{CR}_\bigstar \end{array}$$

To show the soundness of the interpretation w.r.t. the λ-abstraction we will later need the following result:

LEMMA 57 (`Abs_sound`).

$$\left.\begin{array}{r} T \in \mathcal{SN} \\ C \in \mathcal{CR}_\bigstar \\ F \in \mathcal{CR}_{(S\to\bigstar)} \\ \forall N \in C.\ \forall_{\mathcal{ICR}_S} X.\ M[0\backslash N] \in F(X) \end{array}\right\} \Rightarrow \lambda T.M \in \Pi_S(C,F)$$

6.2. Interpretation of terms and types

This is the key of the proof. We build an interpretation where terms (occurring left in judgements) are interpreted by terms, via a parallel substitution, and types (occurring right in judgements) by an invariant candidate.

As usual, we have to give ourselves the interpretation of free variables. Since there are two levels of variables, our interpretation has two components: a term part, and a type part.

DEFINITION 58 (type `intt`). The term part of an interpretation is a function from positive integers to terms. The common notation is Φ.

DEFINITION 59 (`int_term`). The interpretation of a term t in Φ, noted $t[\Phi]$, is the parallel substitution of any free de Bruijn indice i by $\Phi(i)$.

For the predicate level, a smooth way is to use a Σ-type.

DEFINITION 60 (types `Int_K`, `IntP`, a.o.). The interpretation of a variable is either a dependent pair $\langle S, C\rangle$ where $C \in \mathcal{C}_S$, or $\square$. We also consider lists of variable interpretations. Given such a list $\mathcal{I}$, we define $\mathrm{Cl}(\mathcal{I})$ by applying the following mapping to $\mathcal{I}$:

$$\begin{array}{lcl} \langle S, C\rangle & \mapsto & (\mathrm{Knd}\ S) \\ \square & \mapsto & (\mathrm{Typ}\ \bigstar) \end{array}$$

We write $\mathrm{Cl}_\mathcal{I}(T)$ for $\mathrm{Cl}_{\mathrm{Cl}(\mathcal{I})}(T)$. The notions of invariance and extentional equality straightforwardly extend to interpretations.

The interpretation $\langle S, C\rangle$ is intended for predicate variables of skeleton S, and $\Box$ is intended to interpret term variables.

The idea is that to interpret a term variable, we only need a term $(\Phi(n))$, whereas a predicate variable has to be interpreted both by a scheme and a term (the latter to take care of type-redexes).

DEFINITION 61 (`int_typ`) . Let T a term. Given $\mathcal{I}$ and a skeleton S, one defines the type interpretation $[\![T]\!]_{\mathcal{I}}^{S}$ by:

$$\begin{array}{rcl}
[\![s]\!]_{\mathcal{I}}^{S} & = & \mathrm{C}^{\mathrm{d}}_{S} \\
[\![n]\!]_{\mathcal{I}}^{S} & = & C \textsf{ if } \mathcal{I}(n) = \langle S, C\rangle \\
[\![\lambda A.M]\!]_{\mathcal{I}}^{S} & = & \textsf{match } (\mathrm{Cl}_{\mathcal{I}}(A), S) \textsf{ with} \\
 & & \mid\ ((\mathrm{Knd}\ \_), (S_1 \rightarrow S_2)) \mapsto \lambda C.\ [\![M]\!]_{\mathcal{I};\langle S_1, C\rangle}^{S_2} \\
 & & \mid\ ((\mathrm{Typ}\ \_),\ \_) \qquad\qquad\quad \mapsto [\![M]\!]_{\mathcal{I};\Box}^{S} \\
[\![(u\ v)]\!]_{\mathcal{I}}^{S} & = & \textsf{match } \mathrm{Cl}_{\mathcal{I}}(v) \textsf{ with} \\
 & & \mid\ (\mathrm{Typ}\ S_v) \mapsto [\![u]\!]_{\mathcal{I}}^{(S_v \rightarrow S)}([\![v]\!]_{\mathcal{I}}^{S_v}) \\
 & & \mid\ \mathrm{Trm} \qquad\ \ \mapsto [\![u]\!]_{\mathcal{I}}^{S} \\
[\![\Pi T.U]\!]_{\mathcal{I}}^{\star} & = & \textsf{match } \mathrm{Cl}_{\mathcal{I}}(T) \textsf{ with} \\
 & & \mid\ (\mathrm{Knd}\ S_T) \mapsto \Pi_{S_T}([\![T]\!]_{\mathcal{I}}^{\star}, \lambda C.\ [\![U]\!]_{\mathcal{I};\langle S_T, C\rangle}^{\star}) \\
 & & \mid\ \_ \qquad\qquad\ \ \mapsto \Pi_{\star}([\![T]\!]_{\mathcal{I}}^{\star}, \lambda\_.\ [\![U]\!]_{\mathcal{I};\Box}^{\star}) \\
[\![T]\!]_{\mathcal{I}}^{S} & = & \mathrm{C}^{\mathrm{d}}_{S} \textsf{ otherwise}
\end{array}$$

A few remarks: first, this definition is relevant only when T is a well-formed predicate or kind in a context Γ and $S = \mathrm{Typ}^{-1}(\mathrm{Cl}_{\Gamma}(T))$.

It is also interesting to note how the different levels interact. Suppose that $\Gamma \vdash T : K$ and $\Gamma \vdash K : \mathrm{Kind}$. Then, assuming $\mathcal{I}$ and Φ verify some well-chosen conditions to be defined below, we will have:

- The type interpretation of T is a candidate of order K.
- The term interpretation of T is in the type interpretation of K (which is a set of terms).
- Furthermore, if $K = \mathrm{Prop}$, then for any t of type T we also have $t[\Phi] \in [\![T]\!]_{\mathcal{I}}$.

As usual, this definition calls for some stability results.

LEMMA 62 (`int_equiv_int_typ`) . *The interpretations of a given type in two equivalent interpretations are extentionaly equal:*

$$\mathcal{I} \doteq \mathcal{J} \Rightarrow [\![T]\!]_{\mathcal{I}} \overset{S}{=} [\![T]\!]_{\mathcal{J}}.$$

As a consequence, $[\![T]\!]_{\mathcal{I}}$ *is invariant.*

LEMMA 63 (`int_typ_cr`) . *If all variables are interpreted by an invariant candidate, then the interpretation of any type is an invariant candidate.*

$$(\forall n.\ \mathcal{I}(n) = \langle S, C\rangle \Rightarrow C \in \mathcal{ICR}_S) \Rightarrow [\![T]\!]_{\mathcal{I}} \in \mathcal{CR}_S$$

LEMMA 64 (`lift_int_typ`) . *Interpretation is stable by lifting:*

$$(\mathcal{I}; i; \mathcal{I}') \doteq (\mathcal{I}; i; \mathcal{I}') \Rightarrow [\![T]\!]_{\mathcal{I};\mathcal{I}'} \overset{S}{=} [\![\uparrow^1_{|\mathcal{I}'|} T]\!]_{\mathcal{I};i;\mathcal{I}'}$$

The next lemma is usually essential in reducibility proofs. It is interesting to remark that the two results below where, by far, the most tedious of the whole development[1].

LEMMA 65 (`subst_int_typ`) . *Type interpretation is stable (extentionaly) by substitution, provided the substituted variable is correctly interpreted.*

We define the correct possible interpretations inductively:

$$\frac{\mathrm{Cl}_{\mathcal{I}}(M) = (\mathrm{Typ}\ S)}{\left\langle S, [\![M]\!]^S_{\mathcal{I}}\right\rangle \in \mathcal{AVI}_{(M,\mathcal{I})}} \qquad \frac{\mathrm{Cl}_{\mathcal{I}}(M) = \mathrm{Trm}}{\square \in \mathcal{AVI}_{(M,\mathcal{I})}}$$

The following holds:

$$\left.\begin{array}{r} \Gamma \vdash T : K \\ \mathrm{Cl}_\Gamma(T) \neq Trm \\ (\mathcal{I}; i; \mathcal{I}') \doteq (\mathcal{I}; i; \mathcal{I}') \\ \mathrm{Cl}(\Gamma) = \mathrm{Cl}(\mathcal{I}; i; \mathcal{I}') \\ i \in \mathcal{AVI}_{(v,\mathcal{I})} \end{array}\right\} \Rightarrow [\![T]\!]_{\mathcal{I};i;\mathcal{I}'} \overset{\mathrm{Typ}^{-1}(\mathrm{Cl}_\Gamma(T))}{=} [\![T[|\mathcal{I}'|\backslash v]]\!]_{\mathcal{I};\mathcal{I}'}$$

The next lemma is of course needed for soundness w.r.t. the conversion rule.

LEMMA 66 (`conv_int_typ`) . *Type interpretation is (extentionaly) stable by β-conversion.*

$$\left.\begin{array}{r} U \approx_\beta V \\ \mathrm{Cl}_\Gamma(U) \neq Trm \\ \Gamma \vdash U : K \\ \Gamma \vdash V : K \\ \mathcal{I} \doteq \mathcal{I} \\ \mathrm{Cl}(\Gamma) = \mathrm{Cl}(\mathcal{I}) \end{array}\right\} \Rightarrow [\![U]\!]_{\mathcal{I}} \overset{\mathrm{Typ}^{-1}(\mathrm{Cl}_\Gamma(U))}{=} [\![V]\!]_{\mathcal{I}}$$

[1] This is mainly due to the fact that the interpretation is thought of as a partial function and formally defined as a total function. In lots of cases one has to check many conditions in order to determine whether one actually is inside the "interesting domain" or not. In general, it appears that it is very important to chose carefully the values of such functions outside this intended domain (phony values) in order to keep the properties to prove as uniform as possible; this choice can make huge differences in the size of the formal statements and proofs.

6.3. Adapted interpretations

To state the soundness result, we have to restrict quantification to interpretations adapted to the context.

DEFINITION 67 (`int_adapt`) . An interpretation is *adapted* to Γ if it verifies the 3 following conditions:

$$(\mathcal{I},\Phi)\in\mathcal{AI}_\Gamma \overset{\text{def}}{\equiv} \begin{cases} \mathrm{Cl}(\mathcal{I})=\mathrm{Cl}(\Gamma) \\ \forall n.\ \mathcal{I}(n)=\langle S,C\rangle \Rightarrow C\in\mathcal{ICR}_S \\ \forall n.\ \Phi(n)\in[\![\Gamma(n)]\!]^\star_{\mathcal{I}_{|n+1}} \end{cases}$$

We can now state and prove the main result:

THEOREM 68 (`int_sound`) . *For every derivable judgement* $\Gamma\vdash t:T$*, the interpretation of* t *in an interpretation adapted to* Γ *belongs to the interpretation of its type* T*:*

$$\Gamma\vdash t:T\ \wedge\ (\mathcal{I},\Phi)\in\mathcal{AI}_\Gamma\Rightarrow t[\Phi]\in[\![T]\!]^\star_{\mathcal{I}}$$

6.4. The default interpretation

For every well-formed context we produce an adapted interpretation.

DEFINITION 69 (`def_intt`, `def_intp`). The term part of the default interpretation of any variable is itself, and predicate variables are interpreted by the adapted default candidate:

$$\begin{aligned}\Phi^{\mathrm{d}}_\Gamma &= \lambda n.\ n\\ \mathcal{I}^{\mathrm{d}}_{[]} &= [\,]\\ \mathcal{I}^{\mathrm{d}}_{\Gamma;T} &= \begin{cases}\left(\mathcal{I}^{\mathrm{d}}_\Gamma;\left\langle S,\mathrm{C}^{\mathrm{d}}_S\right\rangle\right) & \textsf{if } \mathrm{Cl}_\Gamma(T)=(\mathrm{Knd}\ S)\\ (\mathcal{I}^{\mathrm{d}}_\Gamma;\square) & \textsf{otherwise}\end{cases}\end{aligned}$$

LEMMA 70 (`id_int_term`) . *The term part of the default interpretation is the identity:*

$$t[\Phi^{\mathrm{d}}_\Gamma]=t$$

LEMMA 71 (`def_adapt`) . *The default interpretation is adapted to any well-formed context.*

$$\Gamma\vdash\Rightarrow(\mathcal{I}^{\mathrm{d}}_\Gamma,\Phi^{\mathrm{d}}_\Gamma)\in\mathcal{AI}_\Gamma$$

6.5. Main theorem

We put everything together to get the strong normalization theorem:

THEOREM 72 (`str_norm`, `type_sn`). *All the well-typed terms and inhabited types are strongly normalizing:*

$$\Gamma \vdash t : T \Rightarrow t \in \mathcal{SN} \ \wedge \ T \in \mathcal{SN}$$

7. Checking the programs

We here deal with computational results, i.e. the correctness proofs of the extracted and certified routines. The main point is to prove decidability of type-checking.

We point to [16, 17] for more precise references about program extraction and certification in Coq. Let us just recall that in Coq one distinguishes between *computational* and *non-computational* types. The latter play the role of logical assertions and are erased in the process of program extraction.

For example, the good way to specify a type-checking function is the following formulation of decidability[2]:

```
(e:env)(t,T:term){(typ e t T)}+{~(typ e t T)}
```

The extracted program will take three arguments corresponding to a judgement and return the boolean `true` (corresponding to the left case of disjunction) if the judgement is derivable and `false` if not. Since `typ` is non-computational, the actual derivation will not be built.

The algorithm of the extracted program depends of the structure of the proof of the specification. Using the Program tactic [15] it is possible to use a given program to guide the proof, thus enabling easier control over the obtained algorithm. This feature appeared to be extremely comfortable and well-adapted to the present development.

7.1. Conversion algorithms

It is quite clear and well-known that type-checking a calculus with dependent types requires a conversion check. Of course, this function will be restricted to normalizing terms.

[2] Note that if we were only interested in the correctness, and not completeness, of the implementation, we could use the weaker specification: `(e:env)(t,T:term){(typ e t T)}+{True}`.

The idea is to perform normalization and then check whether the two normal forms are equal. The recursive calls of a normalization algorithm, fed with a term u, take place either on a *strict subterm* of u, or on a reduct of u. In other words, termination of the algorithm depends of the well-foundedness of the following relation:

DEFINITION 73 (predicates `ord_norm1`, `ord_norm`). We define the relation $\prec$ by:

$$x \prec y \overset{\text{def}}{\equiv} (x \subset_{st} y) \vee (y \rhd_\beta x).$$

We call its transitive closure the *normalization order*, written $\prec^+$.

We formulate the termination result re-using definition 9:

THEOREM 74 (`wf_ord_norm`) . *The normalization order $\prec^+$ is well founded for strongly normalizing terms:*

$$\mathcal{SN} \subset Acc_{\prec^+}.$$

PROOF We have already proven $\subset_{st}$ and $\rhd_\beta^{-1}$ commute. Since both are well-founded, this is sufficient for their union to be well-founded (lemma `Acc_union` of the `Coq` theory `RELATIONS/WELLFOUNDED`). ■

LEMMA 75 (program `compute_nf`). *Every strongly normalizing term has a normal form.*

PROOF We used a simple call-by-value strategy. The previous lemma ensures termination (nœtherian recursion over the term to normalize). The only, non-computational results which then have to be proven are quite straightforward:

- The recursive calls follow the normalization order.
- The result of the function is a reduct of the argument and is normal. For instance $\Pi A.B$ is normal if A and B are.

■

COROLLARY 76 (program `is_conv`). *The conversion relation $\approx_\beta$ is decidable on strongly normalizing terms.*

7.2. Auxiliary functions

The following function is used when one has to check that a term is a well-formed type (i.e. is of type a sort).

LEMMA 77 (program `red_to_sort`). *Given any strongly normalizing term T, we can decide whether it is convertible to a sort. If it is, the function returns the sort.*

PROOF We check whether the normal form of T is a sort. ■

To treat the application rule, one has to check whether a type can be converted to a function type.

LEMMA 78 (program `red_to_prod`). *Given a strongly normalizing term T, we can decide whether there exist two terms A and B such that $T \rhd_\beta^* \Pi A.B$. If they exist, the function also returns A and B.*

PROOF We check whether the normal form of T is a product. ■

Note that these two programs could be improved a lot; for instance, reducing T to its weak head normal form would have been sufficient.

7.3. Decidability of type inference and type checking

The main theorem we prove here is the decidability of type-checking. But type-inference is necessary to type-checking, because of the case of application.

We first prove these properties assuming that the context is valid, for efficiency reasons: if we naively follow the inference rules, we will check the validity of the whole context at every leaf of the term. If we assume that the context is valid, we only check validity when the context grows. The gain in complexity is exponential.

THEOREM 79 (program `infer`). *Type inference in a valid context is decidable.*

PROOF The algorithm is based on a recursion over the term to check, which ensures efficiency, except when the conversion rule is often used.

As an example, we only detail the case of the product: to infer the type of $\Pi A.B$, one first infers the type of A and check it to be convertible to a sort. The same is applied to B, but we remember which sort s was convertible with the type of B (`red_to_sort` yields the sort). The type of $\Pi A.B$ is therefore s. Returning the type of B would be wrong, for it is the type of B in the context $(\Gamma; A)$, and it may be ill-typed in Γ. ■

LEMMA 80 (program `check_typ`). *Type-checking in a valid context is decidable.*

LEMMA 81 (program `add_typ`). *For all well-formed context Γ and for all term T, the validity of $\Gamma; T$ is decidable.*

THEOREM 82 (programs `decide_wf`,`decide_typ`). *The context validity and type-checking judgements are decidable in the Calculus of Constructions.*

PROOF We prove the decidability of context validity by induction on the context to check. The inductive case is solved by the program `add_typ`. The second result is a combination of `check_typ` and `decide_wf`. ■

8. Program extraction

Once the essential algorithms were certified, the obvious next step was to use and test them. We therefore implemented a small proof-checker, christened `Coc`, built around the extracted code. The following might be considered as a small-scale model of a user manual for this small-scale proof-checker, annotated with technical details on the implementation.

8.1. Building a stand-alone proof-checker

The idea of the proof-checker is to consider a machine whose state is a set of assumptions, initially empty. This list of assumptions is stored in the global variable `glob_axioms`, and we keep the corresponding print names in `glob_names`. The invariant of the state is the following:

– `glob_axioms` is a well-formed context,

– `glob_axioms` and `glob_names` have the same length,

– all the elements of `glob_names` are different.

One can enter commands to add an axiom, infer the type of a term, or check a typing judgement in this context. The syntax and description of these commands is the following (terms grammar described figure 7):

`Axiom` *ident*:*term*. adds an axiom in the current context. *term* is checked to be a well-formed type and the print name (*ident*) to be fresh.

`Infer` *term*. infers and displays the type of *term* or answers "`mal type`" if the term is ill-typed.

`Check` $term_1$:$term_2$. checks whether $term_2$ is a type of $term_1$, and prints "`Correct`", or "`Echec`" if the judgement doesn't hold.

term :=	*sort*	s
|	*ident*	x
|	`[` *ident-list* `:` *term* `]` *term*	$\lambda\vec{x}{:}t_1.t_2$
|	`(` *term-list* `)`	$(t_1 \ldots t_n)$
|	`let` *ident* `:` *term* `:=` *term* `in` *term*	$(\lambda x{:}t_1.t_3\ t_2)$
|	`(` *ident-list* `:` *term* `)` *term*	$\Pi\vec{x}{:}t_1.t_2$
|	*term* `->` *term*	$t_1 \rightarrow t_2$
sort :=	`Kind`	Kind
|	`Prop`	Prop

Priorities :
- `->` is right associative
- `->` has a higher priority than `[ ]` and `( )`

Figure 7. Coc terms grammar

Note that the programs `add_typ`, `infer`, and `check_typ` of the previous section fit exactly this description. We didn't use the general decidability results, since in our implementation, we build the context incrementally, and we check its validity step by step.

The user interface consists mainly of a parser and a pretty-printer. The commands have several representations, from the most concrete to the most abstract:

1. concrete syntax (string),

2. abstract syntax tree (named variable term),

3. term in de Bruijn notation.

The user interacts with the former, but we formally checked results only on the latter. The translation between 1 and 2 is probably very difficult to formalize and certainly outside the scope of this work. The second translation between 2 and 3 certainly seems feasible and interesting to formalize.

8.2. A complete example in Coc

We can consider Coc as a proof-checker with a very low level mathematical language which precludes using it as a proof assistant.

In [3], Boyer and Dowek explain how a complex proof assistant may be reliable when only a small part of it is certified. One layer is in charge of providing proof-terms, incorporating user-friendly facilities such as subgoal-directed proof search, powerful decision procedures, etc. A second layer (the kernel), reads the proof-term produced by the first layer and answers whether the proof is correct or not. Certifying the kernel ensures the consistency of the whole system.

In order to test Coc, we applied this procedure to an example: Newman's lemma.

Statement of Newman's lemma: if R is a locally confluent and a nœtherian relation, then R is confluent.

We used the formalization of this lemma in the Calculus of Constructions by Huet [12]. In the Coq contributions, the proof appears in the form of tactics. We compiled it in Coq and pretty-printed the resulting λ-terms.

In practice, we had to perform some proof encodings, for Coc doesn't provide constant definition, or the possibility of proving intermediary lemmas. Fully expanding all the constants would obviously lead to a gigantic term. We preferred using two axioms instead; the following Coq definition:

```
Definition x: T := t.
```

would be written in Coc:

```
Axiom x: T.
Axiom unfold_x: (P:T->Prop)(P t)->(P x).
```

Intermediary lemmas can easily be encoded by a β-redex, thanks to the `let in` syntax.

The resulting proof was then given to Coc, which validated it. It is interesting to notice that the extracted code showed up to be surprisingly efficient, faster than the "real" Coq by a factor of 4 (without taking into account Coq's notoriously slow parsing).

This could be improved still further by integrating more carefully this kernel inside a real system, without using the concrete syntax to transfer proof-terms.

9. Conclusion

We view this work as a positive experience. As expected, the way to formalize the definitions was quite straightforward which gives a good confidence about what was actually implemented. Most of the mathematical developments were quite natural and the degree of detail was good: almost all formal results are worth to be informally stated. Note this last claim has to be a little refrained concerning the very first results, but this is probably a subjective impression, since the basic results have, with time, become extremely familiar.

The methodology extracting functional programs from proofs is obviously quite well-suited for the present purpose.

Future work

It seems reasonable to continue the effort, especially by extending the encoded formalism and closing the gap with formalism of Coq itself. We consider the following extensions:

– Definitions: this would require the formalization of δ-reduction, which should be easy.

– A universe mechanism: this would simply require to generalize the proofs to PTSs [2] and should not be difficult (except for normalization).

– Inductive types, which certainly is the crucial and most difficult point, since it induces many complications of the syntax.

– program extraction, which is necessary to fully certify the extracted code.

Achieving this would allow a *bootstrap* of Coq. It does not seem impossible, that such an objective could be within reach in the future.

References

1. Thorsten Altenkirch. A Formalization of the Strong Normalization Proof for System F in LEGO, In *Proceedings of the International Conference on Typed Lamdba Calculi and Applications, TLCA'93.* Springer-Verlag, LNCS 664, March 1993.
2. H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Vol II, Elsevier, 1992.
3. Robert S. Boyer, Gilles Dowek. Towards Checking Proof-Checkers. In Herman Geuvers, editor, *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs*, May 1993.
4. N.J. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Math. Vol. 34 (5), pp. 381–392*, 1972.
5. Thierry Coquand. Une Théorie des Constructions. Thèse de doctorat, Université Paris 7, 1985.
6. Thierry Coquand, Gérard Huet. The Calculus of Constructions, in: *Information and Computation Vol. 76, February/March 1988* (ed.A.R.Meyer), Academic Press, London, 95-120.

7. Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saïbi, Benjamin Werner. The Coq Proof Assistant Reference Manual Version 5.10. Rapport Technique 0177. Projet Coq-INRIA Rocquencourt-ENS Lyon. Juillet 95.
8. J. H. Geuvers. A short and flexible proof of strong normalization for the Calculus of Constructions. In *Types for Proofs and Programs*, P. Dybjer, B. Nordström and J. Smith (eds), LNCS 996, Springer, 1994.
9. Herman Geuvers, Mark-Jan Nederhof. A Modular Proof of Strong Normalization for the Calculus of Constructions. *Journal of Functional Programming*, 1(2):155-189, April 1991.
10. J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Univerité Paris 7, 1972.
11. J.-Y. Girard, Y. Lafont, P. Taylor. Proofs ans Types. *Cambridge Tracts in Theoretical Computer Science 7*. Cambridge University Press.
12. Gérard Huet. The Constructive Engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. WorldScientific Publishing, 1989. Commemorative Volume for Gift Siromoney.
13. Gérard Huet. Residual Theory in λ-calculus: A Complete Gallina Development. Rapport de recherche INRIA 2002, 1993.
14. B. Nordström. Terminating General Recursion. *BIT*, 28, 1988.
15. Catherine Parent. Developing certified programs in Coq – The Program tactic. In *Types for Proofs and Programs '93*, LNCS 806, Springer, 1993.
16. Christine Paulin-Mohring. Extraction de programmes dans le Calcul des Constructions. Thèse de doctorat, Université Paris 7, 1989.
17. Christine Paulin-Mohring, Benjamin Werner. Synthesis of ML programs in Coq. *Journal of Symbolic Computation–special issue on automated programming*,1993.
18. Robert Pollack. A Proof Checker for the Extended Calculus of Constructions. Ph. D. Thesis, University of Edinburgh, 1994.

Address for correspondence: Bruno.Barras, Benjamin.Werner@inria.fr