Verification of the Interface of a Small Proof System in Coq

Bruno Barras

INRIA-Rocquencourt
Domaine de Voluceau, B.P. 105,
78153 Le Chesnay Cedex, France

Abstract. This article describes the formalization of the interface of a proof-checker. The latter is based on a kernel consisting of type-checking functions for the Calculus of Constructions, but it seems the ideas can generalize to other type systems, as far as they are based on the proofs-as-terms principle. We suppose that the metatheory of the corresponding type system is proved (up to type decidability). We specify and certify the toplevel loop, the system invariant, and the error messages.

1 Introduction

Our ultimate goal is to verify a whole proof system. This would improve the reliability of the tool, but it would also be a good benchmark for the methodology of formally certified software design Coq [4] recommends. Applying this to the proof system itself would eventually lead to a *bootstrapped* version of Coq.

One of the major contributions of the LCF system [12], is that it implements a reliable proof-checker with a very small part of critical code: the *kernel*. Around this kernel stands an interface, in charge of translating the high-level directives of the user (such as tactics), into a proof-term that the kernel is able to check. The interface does not need to be certified because the kernel re-checks everything and would refuse a wrong proof. This approach makes it easier to formally check the program.

Following the presentation of cross-checking in [5], a proof-checker for the Calculus of Construction was certified [2, 3], requiring the formalization of the metatheory of this type system. This work has to be related to previous ones, such as [18, 19]. The extraction (some details on the extraction in Coq may be found in [16]) of the type-checking functions was performed, and a hand-written interface allowed to use it like a stand-alone proof system.

It appeared to be efficient, but unable to help the user when developing a proof, since it supports no error messages. Another drawback is that it allows only a linear presentation, proving one lemma after another. As a motivation, an interesting feature of a proof assistant the above system cannot provide, is to allow a modular and parameterized style. When implementing a proof system, there are two ways to have this kind of feature supported:

– By extending the type system: adding a module system with functors would allow parametric developments. This solution surely is the best in the long

term, but it requires the extension to be fully understood and formalized, which is not the case for many advanced features. It is not considered in the present article, but some recent works [14, 9] develop a metatheory of module systems.
- A preprocessing function may compile the higher-level language into the basic operations, given to the kernel. This approach is illustrated by the section mechanism of `Coq` (see example below).

The latter solution is easier to implement, but may jeopardize the soundness of the system, because it does not only affect the way to produce proof-terms, but also the way the theorems are stated. The problem is not that we may prove absurd propositions (this is guaranteed by the typing rules). It is that we are not sure that what the kernel actually checked is what we meant. The user has to check that the statements were processed as he expected. Indeed, if we compare this preprocessing function to a compiler, it is not very satisfying to ask the user to check the compiled code.

The following example illustrates this idea that soundness is not just a matter of typing. In `Coq`, the status of axioms inside a section is very sensitive: depending on the keyword that introduced it, it is either considered as a local hypothesis to be discharged at the end of the section, or an axiom that should survive the section closure.

```
Section VariableOrParameter.
  Variable absurd: False.

  Lemma false_impl_0_equal_1: O=(S O).
  Proof (False_ind O=(S O) absurd).
End VariableOrParameter.
```

When exiting the section, the axiom `absurd` is discharged: `absurd` disappears from the context, and the lemma `false_impl_0_equal_1` has the type `False->O=(S O)`. If we replace the word "`Variable`" by "`Parameter`", the script has radically different semantics: the two absurd statements `False` and `O=(S O)` are left unchanged after the section `S` is closed, which is undoubtedly not what the user expects. If he does not check the exact state of the environment, he may believe he proved `O=(S O)`. This is, of course, not the case.

The fact that the semantics rely on the choice of one word may be confusing for the user. From the the implementation point of view, this may lead to numerous subtle mistakes. We are thus urged to urged to formalize a command language. In this article, we consider three different components in a proof-checker:

- the parser and pretty-printer, reading commands on the keyboard and displaying messages on the screen,
- the interface with the user, the link between the command language and the typing operations to be performed,
- the kernel which provides the type-checking functions.

We describe the formalization[1] of the last two parts for a proof system based on the Calculus of Constructions: a basic command language with error messages, but without section mechanism. This language being defined and explained in the user manual, we feel that a good idea is to follow the style of this document, i.e. describing the commands by giving their effect on the system state, the list of errors they may raise, and what these errors mean.

2 Kernel

This section gives a quick description of the Calculus of Constructions (introduced by Coquand and Huet in [7]), and the metatheoretic results upon which we have built the proof system. The following can be read as a kind of interface specification for the kernel.

The internal representation of sorts, terms and contexts in de Bruijn notations uses the following syntax (terms and types share the same syntactic class):

$$\begin{aligned} \textsc{Srt} &:= \star \mid \square \\ \textsc{Trm} &:= s \mid n \mid \lambda T_1.T_2 \mid (T_1\ T_2) \mid \Pi T_1.T_2 \\ \textsc{Ctx} &:= [\,] \mid \Gamma; T \end{aligned}$$

where $s \in \textsc{Srt}$, $T, T_1, T_2 \in \textsc{Trm}$, $\Gamma \in \textsc{Ctx}$ and $n \in \mathbb{N}$. We write $|t|$ for the highest free de Bruijn variable of t.

Contexts are lists of terms. The list notations used in this article are $|\Gamma|$ for the length of lists, and $\Gamma(n)$ for the n-th item (the rightmost item has rank 0).

The one step β-reduction is written $\rhd_\beta$. Its reflexive transitive closure is $\rhd_\beta^*$, and $\approx_\beta$ stands for the β-conversion.

Then we define two forms of judgements:

- a context validity judgement, noted $\Gamma \vdash$, which expresses that the assumptions in Γ are well-formed,
- a typing judgement, noted $\Gamma \vdash m : t$, expressing that the term m has type t in the context Γ.

The precise inference rules (Fig. 1) for these judgements are not really relevant to understand the discussion: this could be adapted to the case of Pure Type Systems (PTS) [1]. Let us just stress that they are mutually recursive: terms are typed only in valid contexts, and the validity of a context requires that all its components are typed by sorts.

We prove the usual metatheoretic properties of type systems as in [11]. A detailed description of the proofs formalized in `Coq` can be found in [2, 3]. We only list the main properties:

- Confluence of β-reduction: the main corollary is the unicity of normal forms

$$a \rhd_\beta^* b \ \wedge\ a \rhd_\beta^* c \ \Rightarrow\ \exists d.\ b \rhd_\beta^* d \ \wedge\ c \rhd_\beta^* d$$

[1] The whole proof script is available at the following address: `http://pauillac.inria.fr/~barras/typechecker/`

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

$$\text{(PROP)}\ \frac{\Gamma\vdash}{\Gamma\vdash\star : \Box} \qquad \text{(VAR)}\ \frac{\Gamma\vdash \qquad \uparrow^{n+1}\Gamma(n)=T}{\Gamma\vdash n : T}$$

$$\text{(ABS)}\ \frac{\Gamma\vdash T : s_1 \qquad \Gamma;T\vdash M : U \qquad \Gamma;T\vdash U : s_2}{\Gamma\vdash \lambda T.M : \Pi T.U}(s_1,s_2\in\mathrm{SRT})$$

$$\text{(APP)}\ \frac{\Gamma\vdash v : V \qquad \Gamma\vdash u : \Pi V.T}{\Gamma\vdash (u\ v) : T[0:=v]}$$

$$\text{(PROD)}\ \frac{\Gamma\vdash T : s_1 \qquad \Gamma;T\vdash U : s_2}{\Gamma\vdash (\Pi T.U) : s_2}\ (s_1,s_2\in\mathrm{SRT})$$

$$\text{(CONV)}\ \frac{\Gamma\vdash M : U \qquad \Gamma\vdash V : s \qquad U\approx_\beta V}{\Gamma\vdash M : V}\ (s\in\mathrm{SRT})$$

Fig. 1. Typing Rules for the Calculus of Constructions

– Subject reduction: a term keeps its type by reduction

$$\Gamma\vdash m : t \ \wedge\ m \rhd_\beta^* m' \ \Rightarrow\ \Gamma\vdash m' : t$$

– Type uniqueness modulo β-conversion: all the types of a given term are convertible

$$\Gamma\vdash m : u \ \wedge\ \Gamma\vdash m : v \ \Rightarrow\ u\approx_\beta v$$

– Strong normalization ($\mathcal{SN}$ is the set of terms from which no infinite reduction path starts): all the well-typed terms are strongly normalizing

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

– Decidability of typing judgements:

$$(\Gamma\vdash)+\neg(\Gamma\vdash)$$
$$(\Gamma\vdash m : t)+\neg(\Gamma\vdash m : t)$$

The $+$ denotes the constructive OR connective; this means we can produce by extraction a program that answers `true` when the judgement is derivable, and `false` otherwise. We will also write $\exists^* x.\ P(x)$ for the constructive existential, which gives, by extraction, a program computing a witness x satisfying the predicate P.

We could have proved simultaneously the decidability results by a mutual recursion, but this would yield a very inefficient program, which would re-check the validity of the context a huge number of times[2]. The resulting exponential complexity factor is avoided, by distinguishing three typing operations:

- type-inference in a valid context

$$\Gamma \vdash \Rightarrow\ \exists^* t.\ (\Gamma \vdash m : t)\ +\ \forall t.\ \neg(\Gamma \vdash m : t) \tag{1}$$

- type-checking in a valid context

$$\Gamma \vdash \Rightarrow\ (\Gamma \vdash m : t) + \neg(\Gamma \vdash m : t) \tag{2}$$

- checking the extension of a valid context

$$\Gamma \vdash \Rightarrow\ (\Gamma; t \vdash) + \neg(\Gamma; t \vdash) \tag{3}$$

The decidability results can easily be proved: $(\Gamma \vdash) + \neg(\Gamma \vdash)$, by repeating (3) on the successive terms of Γ, and $(\Gamma \vdash m : t) + \neg(\Gamma \vdash m : t)$, by checking Γ, then typechecking m with (2).

We give no details on the type-checking algorithm. It should not be necessary, because we describe a general method we want to apply to many type systems. The question of finding algorithms for type systems is a major concern in type theory, and has been answered for various systems by [15, 13, 18, 2] and many others. The above functions may take advantage of all these works. As an example, we could consider the type-checking function as a strategy for the Constructive Engine of Huet [13].

Since a proof may be reduced to a derivable typing judgement $\Gamma \vdash m : t$ stating that under the assumptions Γ, m is a proof of the proposition t, this may be already considered as a proof-checker. But it cannot be considered as a proof system, because it provides no help in error recovery, and the judgement has to be given in a whole. In the following sections, we describe and formalize the interface, that transforms a mere type-checker into an executable proof system.

3 Certified Error Messages

The first feature we add is the possibility to return error messages instead of an uninformative failure. These messages should help understanding what is wrong, and should give the location of the error. In our approach, it is not just a comment that is printed when one of the typing operations fails; it should carry a piece of *checked* information that explains why the typing operation cannot succeed.

Error messages are usually directly displayed as strings. But it would be obviously tedious and awkward to have the main type-checking loop handle actual

[2] This is similar to the naive way of computing the terms of Fibonacci's sequence, with a double recursive call. We use the same trick, keeping track of the context validity proofs. Note that, by extraction, these proofs are not actually computed.

messages in string format. Instead, we define an abstract notation, corresponding to the different possible failures of the typing process:

$$\begin{aligned}\text{TYPERR} := \; & \text{Under}(T, e) \mid \text{DbError}(n) \mid \text{Topsort}(\text{s}) \\ & \mid \text{LambdaTopsort}(T, s) \mid \text{ExpectType}(T_1, T_2, T_3) \\ & \mid \text{NotAType}(T_1, T_2) \mid \text{NotAFun}(T_1, T_2) \\ & \mid \text{ApplyErr}(T_1, T_2, T_3, T_4)\end{aligned}$$

where $T, T_1, T_2, T_3, T_4 \in \text{TRM}$, $e \in \text{TYPERR}$, $s \in \text{SRT}$ and $n \in \mathbb{N}$.

These abstract syntax trees have a concrete representation as strings. This translation is straightforward. As an example, Fig. 2 shows how they would be mapped in an English language-tuned implementation.

Under($t_1, \ldots$ Under(t_n, e))	"In the local context $t_1, \ldots, t_n$, e"
DbError(n)	"The de Bruijn variable n is unbound"
Topsort(s)	"The sort s is ill-typed"
LambdaTopsort(t, s)	"The term t is a λ-abstraction over a type of sort s"
ExpectType(m, t, t_e)	"The term m of type t, is expected to have the type t_e"
NotAType(m, t)	"The term m of type t cannot be typed by a sort"
NotAFun(m, t)	"The term m of type t is non-functional"
ApplyErr(u, a, v, b)	"The term u of type a cannot be applied to v of type b"

Fig. 2. Concrete syntax of error messages

Note that Under(T, e) is the only recursive constructor. It contains the type of the variables of the local context the error occurred in. It could be chosen more verbose and keep track of the exact path in the term, and not only the successive local variables.

Error messages depend on the type system: for the general case of PTS, we would need an error message for forbidden product formations; for those not satisfying type uniqueness, we may want to infer the principal type, etc. But we claim that the method remains general enough to be applied in other cases. Adapting the error messages to the type system is not the main point here.

The messages must use a language the user can understand, regardless of the actual type-checking algorithm. All we require is the knowledge of the typing rules and the basic metatheoretic results[3] such as subject-reduction, the fact that objects of type $\Box$ cannot be abstracted, etc.

We define Expln_Γ, the set of meaningful errors in context Γ, given as Prolog-like clauses in Fig. 3. In Coq, we use an inductive predicate, well adapted to this

[3] The knowledge necessary for PTS is not so important. For ML, understanding error messages requires some notions of unification, which is much more difficult.

kind of definition. This predicate is meant to encode the meaning of the error messages by a logical proposition. We are ensured that if the error e belongs to Expln_Γ, then there is a proof of what the error message intuitively means (compare Fig. 2 and 3). Let us explain two of these clauses:

- In order to be typed, a de Bruijn variable n must be defined in (or bound by) the context, i.e. n must be less than the length of the context. Thus, the error $\mathrm{DbError}(n)$ may be raised when n is greater than or equal to the length of the context, and the variable is said unbound.
- The error message $\mathrm{ExpectType}(m, t_{\mathrm{act}}, t_{\mathrm{exp}})$ is said consistent only when we have a proof that $\Gamma \vdash m : t_{\mathrm{act}}$ is derivable but not $\Gamma \vdash m : t_{\mathrm{exp}}$, which is the formal equivalent of the associated sentence in natural language. t_{exp} may be ill-typed, but all its variables must be bound in the context, or else it could not be printed.

$$\frac{\Gamma; t \vdash \qquad e \in \mathrm{Expln}_{\Gamma;t}}{\mathrm{Under}(t, e) \in \mathrm{Expln}_\Gamma} \qquad \frac{n \geq |\Gamma|}{\mathrm{DbError}(n) \in \mathrm{Expln}_\Gamma}$$

$$\frac{\forall t.\ \neg(\Gamma \vdash s : t)}{\mathrm{Topsort(s)} \in \mathrm{Expln}_\Gamma} \qquad \frac{\Gamma; t \vdash m : s \qquad \forall t.\ \neg(\Gamma \vdash s : t)}{\mathrm{LambdaTopsort}(\lambda t.m, s) \in \mathrm{Expln}_\Gamma}$$

$$\frac{\Gamma \vdash m : t_{\mathrm{act}} \qquad \neg(\Gamma \vdash m : t_{\mathrm{exp}}) \qquad |t_{exp}| < |\Gamma|}{\mathrm{ExpectType}(m, t_{\mathrm{act}}, t_{\mathrm{exp}}) \in \mathrm{Expln}_\Gamma}$$

$$\frac{\Gamma \vdash m : t \qquad \forall s \in \textsc{Srt}.\ \neg(\Gamma \vdash m : s)}{\mathrm{NotAType}(m, t) \in \mathrm{Expln}_\Gamma}$$

$$\frac{\Gamma \vdash m : t \qquad \forall a, b.\ \neg(\Gamma \vdash m : \Pi a.b)}{\mathrm{NotAFun}(m, t) \in \mathrm{Expln}_\Gamma}$$

$$\frac{\Gamma \vdash u : \Pi a.b \qquad \Gamma \vdash v : c \qquad \neg(\Gamma \vdash v : a)}{\mathrm{ApplyErr}(u, \Pi a.b, v, c) \in \mathrm{Expln}_\Gamma}$$

Fig. 3. Meaning of errors

This notion discards inconsistent error messages such as "$\star$ isn't typed by a sort" (its type is the sort $\square$), or the very annoying "m has type t, but is expected to have type t", sometimes yielded by bogus compilers.

Being more precise, only a subset of all possible errors can be raised for each of the three possible typing operations:

- The set InfErr_t gives the allowed errors when inferring the type of t. It is recursively defined:

$$\mathrm{InfErr}_s = \{\mathrm{Topsort(s)}\}$$

$$\begin{aligned}
\mathrm{InfErr}_n &= \{\mathrm{DbError}(n)\}\\
\mathrm{InfErr}_{\lambda t.m} &= \mathrm{InfErr}_t \ \cup\ \mathrm{NotAType}(t, \textsc{Trm}) \ \cup\ \mathrm{Under}(t, \mathrm{InfErr}_m)\\
&\quad \cup\ \{\mathrm{LambdaTopsort}(\lambda t.m, s)\}\\
\mathrm{InfErr}_{(u\ v)} &= \mathrm{InfErr}_u \ \cup\ \mathrm{NotAFun}(u, \textsc{Trm}) \ \cup\ \mathrm{InfErr}_v\\
&\quad \cup \mathrm{ApplyErr}(u, \textsc{Trm}, v, \textsc{Trm})\\
\mathrm{InfErr}_{\Pi t.u} &= \mathrm{InfErr}_t \ \cup\ \mathrm{NotAType}(t, \textsc{Trm}) \ \cup\ \mathrm{Under}(t, \mathrm{InfErr}_u)\\
&\quad \cup \mathrm{Under}(t, \mathrm{NotAType}(u, \textsc{Trm}))
\end{aligned}$$

We prove that any meaningful error of InfErr_t witnesses that t is an ill-typed term (i.e. the inference operation fails):

$$e \in \mathrm{Expln}_\Gamma \cap \mathrm{InfErr}_m \Rightarrow \forall t.\ \neg(\Gamma \vdash m : t) \tag{4}$$

– Checking whether m has type t fails when m is ill-typed, t is ill-typed (but not equal to $\Box$) or when the inferred type of m is not convertible with t:

$$\begin{aligned}
\mathrm{ChkErr}_{(m:t)} = \mathrm{InfErr}_m \ \cup\ &\{e \mid e \in \mathrm{InfErr}_t \ \wedge\ t \neq \Box\}\\
&\cup \mathrm{ExpectType}(m, \textsc{Trm}, t)
\end{aligned}$$

Here again, the restriction to the set $\mathrm{ChkErr}_{(m:t)}$ allows to prove that m has not the type t:

$$e \in \mathrm{Expln}_\Gamma \cap \mathrm{ChkErr}_{(m:t)} \Rightarrow \neg(\Gamma \vdash m : t) \tag{5}$$

– Adding a variable of type t is forbidden when t is ill-typed or when it is not typed by a sort:

$$\mathrm{DeclErr}_t = \mathrm{InfErr}_t \ \cup\ \mathrm{NotAType}(t, \textsc{Trm})$$

The meaningful errors of $\mathrm{DeclErr}_t$ implies that $\Gamma; t$ is not valid:

$$e \in \mathrm{Expln}_\Gamma \cap \mathrm{DeclErr}_t \Rightarrow \neg(\Gamma; t \vdash) \tag{6}$$

The following examples illustrate the necessity of these subsets: when trying to infer the type of $\lambda\star.\lambda 0.0$[4], we do not allow any error to be raised because this term is well-formed: its type is $\Pi\star.\Pi 0.1$[5]. We avoid completely unrelated error messages such as DbError(42657) (the de Bruijn 42657 does not occur in the term), but also the irrelevant, though consistent, errors such as $\mathrm{NotAType}(\lambda\star.\lambda 0.0, \Pi\star.\Pi 0.1)$, because pointing out that a term is not typed by a sort does not imply that it has no type.

In the present work, error messages reflect the steps of the type-checking function. We propose a variation on the above definition, to show that this similarity is not necessary even though it makes the proofs easier. We could define InfErr_m and $\mathrm{ChkErr}_{(m:t)}$ as mutually dependent sets: type-checking involves

[4] This term would be written $\lambda A\!:\!\star.\lambda x\!:\!A.x$ in named variable notation

[5] stands for $\Pi A\!:\!\star.\Pi x\!:\!A.A$

type-inference and type-inference involves type-checking in the case of the application. When inferring the type of $(u\ v)$ with u of type $\Pi a.b$, we check that v has type a. The type-checking function could return the error $\mathrm{ExpectType}(v, t, a)$, but it is more appropriate to return the error $\mathrm{ApplyErr}(u, \Pi a.b, v, t)$ of our specification, because it gives more details on the context of the error.

These error messages are incorporated in the typing functions (i.e. in the kernel). This gives raise to the following redesigned specifications:

$$\Gamma \vdash \Rightarrow\ \exists^* t.\ (\Gamma \vdash m : t) + \exists^* e \in \mathrm{Expln}_\Gamma \cap \mathrm{InfErr}_m$$
$$\Gamma \vdash \Rightarrow\ (\Gamma \vdash m : t) + \exists^* e \in \mathrm{Expln}_\Gamma \cap \mathrm{ChkErr}_{(m:t)}$$
$$\Gamma \vdash \Rightarrow\ (\Gamma; t \vdash) + \exists^* e \in \mathrm{Expln}_\Gamma \cap \mathrm{DeclErr}_t$$

The algorithms used are the same, but instead of just proving derivability or not of a given judgement, we give a certified error message with its justification when typing fails. Remark that these specifications are a refinement of the original ones, thanks to the results (4, 5, 6).

The error messages appear as an extension of the kernel, but now, it is possible to have an algorithm without error messages. If we consider that errors are indeed a kind of debugging facility, we are not very concerned by reporting errors efficiently. In that case, we can use any algorithm returning uninformative failure, and if it fails, call our algorithm that will also fail, but with informative failure.

4 Specification of the Machine

We would like to integrate the kernel in a proof system allowing the development of proofs in a more natural way for the user (i.e. closer to the usual mathematical style). In this section, we show how the incremental style of the mathematical language may be reflected.

To any point of a theory script, we can associate the list of assumptions made. Every sentence of the script refers implicitly to this list, and can modify it. This incremental style recalls the notion of state. Therefore, we model the proof system as an automaton whose state is mainly a list of axioms. The state of the system may change by transitions, enabled by reading commands in an input stream. There are two kinds of transitions:

- successful transitions, producing an acknowledgment message and the new state of the system,
- error transitions, yielding an error message.

The state of the proof-checker is a well-formed context, with the list of print names associated to these axioms. The print names should be unique and the two lists have the same length. This is formalized by a structure $S = (\Gamma, \Sigma, \pi_1, \pi_2, \pi_3)$

composed of two computational objects and three proofs (the latter are sometimes omitted and a state may be noted (Γ, Σ)), such that:

$$\begin{aligned} &\Gamma \in \textsc{Ctx} \\ &\Sigma \in \textsc{Name}^* \\ \pi_1 &: \Gamma \vdash \\ \pi_2 &: |\Gamma| = |\Sigma| \\ \pi_3 &: \mathrm{AllDiff}(\Sigma) \end{aligned}$$

The set of names (Name) is abstract. For the moment, we only require the equality of it to be decidable. $\textsc{Name}^*$ is the set of lists of names, and the proposition $\mathrm{AllDiff}(\Sigma)$ means that all the components of Σ are distinct:

$$\mathrm{AllDiff}(\Sigma) \ := \ \forall i, j.\ \Sigma(i) = \Sigma(j) \Rightarrow i = j$$

The fields π_1, π_2, π_3 are the proofs of the invariant of the system. Since the proofs are part of the structure, the invariant is guaranteed by typing. The proof π_2 ensures that every axiom has a name, and π_3 avoids hiding a variable by declaring another one with the same name.

The operations of that machine are similar to the ones the user will enter. As a consequence, a machine like the Constructive Engine is not what we need, because it can be used only to type-check terms. We are enclosing the type-checking machines (or functions) inside a new one. The set of commands defines the mathematical language of the proof system. We consider a very simple one, consisting of six commands:

$$\begin{aligned} \textsc{Command} := \ & \mathrm{INFER}(T) \mid \mathrm{CHECK}(T_1, T_2) \mid \mathrm{AXIOM}(x, T) \\ & \mid \mathrm{DELETE} \mid \mathrm{LIST} \mid \mathrm{QUIT} \end{aligned}$$

performing, respectively, type inference of a term, checking that a term has a given type, adding an assumption to the current state, removing the latest introduced axiom, printing the list of current assumption names, and exiting.

For each successful command, the proof system answers with one of the following messages (like for error messages, there is in each case, a corresponding natural language sentence):

$$\begin{aligned} \textsc{Message} := \ & \mathrm{Inferred}(T) \mid \mathrm{Correct} \mid \mathrm{Assumed}(x) \mid \mathrm{Deleted}(x) \\ & \mid \mathrm{DisplayNames}(\Sigma) \mid \mathrm{Exiting} \end{aligned}$$

A predicate defines the correct transitions: the rules in Fig. 4 define the proposition $(S_\mathrm{i}, c) \xrightarrow{\mathrm{exec}} (S_\mathrm{f}, m)$, meaning that in the state S_i, the command c allows to reach the state S_f, and the acknowledgment message m is displayed.

Note that in the axiom rule, we do not have to specify that $\Gamma; T$ is a valid context and x is fresh, because this is ensured by the invariant. This allows concise (and more readable) specifications. The premises of the first two rules represent calls to auxiliary machines beside those required to maintain the invariant. They would not be necessary if any derivable judgement (context validity or typing

$$\frac{\Gamma \vdash m : t}{((\Gamma, \Sigma), \mathrm{INFER}(m)) \xrightarrow{\mathrm{exec}} ((\Gamma, \Sigma), \mathrm{Inferred}(t))}$$

$$\frac{\Gamma \vdash m : t}{((\Gamma, \Sigma), \mathrm{CHECK}(m, t)) \xrightarrow{\mathrm{exec}} ((\Gamma, \Sigma), \mathrm{Correct})}$$

$$\frac{}{((\Gamma, \Sigma), \mathrm{AXIOM}(x, t)) \xrightarrow{\mathrm{exec}} (((\Gamma; t), (\Sigma; x)), \mathrm{Assumed}(x))}$$

$$\frac{}{(((\Gamma; t), (\Sigma; x)), \mathrm{DELETE}) \xrightarrow{\mathrm{exec}} ((\Gamma, \Sigma), \mathrm{Deleted}(x))}$$

$$\frac{}{((\Gamma, \Sigma), \mathrm{LIST}) \xrightarrow{\mathrm{exec}} ((\Gamma, \Sigma), \mathrm{DisplayNames}(\Sigma))}$$

$$\frac{}{((\Gamma, \Sigma), \mathrm{QUIT}) \xrightarrow{\mathrm{exec}} ((\Gamma, \Sigma), \mathrm{Exiting})}$$

Fig. 4. Successful transitions

judgement) could be the state of the proof-checker. But it would split commands like INFER and CHECK in two subcommands: one building the judgement, and the other one displaying and discarding the judgement, thus restoring the initial state. Also, we consider these two commands as queries about the current state. Therefore, they should not change the state. Only AXIOM and DELETE should.

When no successful transition can occur, the system may return an error message. The latter are not only the type error messages introduced in the previous section: some errors may result from a bad usage of commands that would break the invariant. The set of errors is the following:

$$\textsc{Error} := \mathrm{NameClash}(x) \mid \mathrm{TypeError}(e) \mid \mathrm{CantDelete}$$

where $x \in \textsc{Name}$ and $e \in \textsc{TypErr}$.

The error $\mathrm{NameClash}(x)$ is raised when a command tries to add an axiom under the already used name x. The command DELETE raises CantDelete when the current context is empty. These error transitions are formally defined in Fig. 5.

The main result is that for any command in any state, we can either reach a new state printing an acknowledgment message, or find an error consistent with the initial state:

$$\forall S_{\mathrm{i}}, c.\ \exists^* (S_{\mathrm{f}}, m).\ (S_{\mathrm{i}}, c) \xrightarrow{\mathrm{exec}} (S_{\mathrm{f}}, m)\ +\ \exists^* e.\ (S_{\mathrm{i}}, c) \xrightarrow{\mathrm{error}} e$$

This ensures that the extracted program `interp_command` (the command interpreter) is sound, i.e. it allows only valid transitions.

$$\frac{e \in \mathrm{Expln}_{\Gamma} \cap \mathrm{InfErr}_m}{((\Gamma, \Sigma), \mathrm{INFER}(m)) \stackrel{\text{error}}{\longrightarrow} \mathrm{TypeError}(e)}$$

$$\frac{e \in \mathrm{Expln}_{\Gamma} \cap \mathrm{ChkErr}_{(m:t)}}{((\Gamma, \Sigma), \mathrm{CHECK}(m, t)) \stackrel{\text{error}}{\longrightarrow} \mathrm{TypeError}(e)}$$

$$\frac{e \in \mathrm{Expln}_{\Gamma} \cap \mathrm{DeclErr}_t}{((\Gamma, \Sigma), \mathrm{AXIOM}(x, t)) \stackrel{\text{error}}{\longrightarrow} \mathrm{TypeError}(e)}$$

$$\frac{x \in \Sigma}{((\Gamma, \Sigma), \mathrm{AXIOM}(x, t)) \stackrel{\text{error}}{\longrightarrow} \mathrm{NameClash}(x)}$$

$$\frac{}{(([\,], \Sigma), \mathrm{DELETE}) \stackrel{\text{error}}{\longrightarrow} \mathrm{CantDelete}}$$

Fig. 5. Error transitions

The following lemma is a kind of completeness result:

$$\neg(\ (S_{\mathrm{i}}, c) \stackrel{\text{exec}}{\longrightarrow} (S_{\mathrm{f}}, m) \quad \wedge \quad (S_{\mathrm{i}}, c) \stackrel{\text{error}}{\longrightarrow} e\ )$$

It proves that in a given state, an error may be raised only when there is no transition towards a new state. Therefore the system will never reject well-formed commands.

5 De Bruijn vs Named Variables Notations

Type systems are generally easier to formalize in de Bruijn notation [6] because it avoids the renaming problems, and gives better intuition on the scope of variables. But this notation isn't user-friendly and the human wants to handle names.

This section describes how this translation between named variables and de Bruijn notation can be made, by adding a layer on top of the proof system of the previous section.

The abstract type of names of the previous section may be instantiated by any type fulfilling some basic conditions. The equality of names should be decidable, as already required, but we also require the existence of an injection of naturals to names:

$$\exists^* f \in \mathbb{N} \rightarrow \textsc{Name}.\ \forall n, m \in \mathbb{N}.\ f(n) = f(m) \Rightarrow n = m$$

If we choose the type `string` of Objective Caml to implement the set of names, the function `(fun n -> "x"^(string_of_int n))` fits (ignoring overflow problems). It would be equivalent to assume that the set of names is infinite:

$$\forall \Sigma \in \textsc{Name}^*.\ \exists^* x \notin \Sigma$$

Indeed, the above statement specifies a function that, given a list of names, returns a fresh name.

The parser returns expressions with names:

$$\textsc{Expr} := s \mid x \mid \lambda x : E_1 . E_2 \mid (E_1\ E_2) \mid \Pi x : E_1 . E_2$$

where $s \in \textsc{Srt}$, $x \in \textsc{Name}$ and $E_1, E_2 \in \textsc{Expr}$. We define $\mathrm{FV}(E)$, the set of free variables of an expression E, in the usual way.

Given a list of print names Σ, we define the translation predicate, Fig. 6. The n-th item of the list is the name of the de Bruijn variable n. We cannot assume all the elements of Σ to be distinct, because we must allow the hiding of variables, as in $\lambda x : A . \lambda x : B . x$. Instead, we can translate a de Bruijn variable n into x only when the latest occurrence of x in Σ is at rank n.

$$\frac{}{\Sigma \vdash s \xleftrightarrow{\mathrm{dB}} s}\ (s \in \textsc{Srt})$$

$$\frac{\Sigma(n) = x \qquad \forall k < n . \Sigma(k) \neq x}{\Sigma \vdash x \xleftrightarrow{\mathrm{dB}} n}$$

$$\frac{\Sigma \vdash E_1 \xleftrightarrow{\mathrm{dB}} A \qquad \Sigma; x \vdash E_2 \xleftrightarrow{\mathrm{dB}} M}{\Sigma \vdash \lambda x : E_1 . E_2 \xleftrightarrow{\mathrm{dB}} \lambda A . M}$$

$$\frac{\Sigma \vdash E_1 \xleftrightarrow{\mathrm{dB}} u \qquad \Sigma \vdash E_2 \xleftrightarrow{\mathrm{dB}} v}{\Sigma \vdash (E_1\ E_2) \xleftrightarrow{\mathrm{dB}} (u\ v)}$$

$$\frac{\Sigma \vdash E_1 \xleftrightarrow{\mathrm{dB}} A \qquad \Sigma; x \vdash E_2 \xleftrightarrow{\mathrm{dB}} M}{\Sigma \vdash \Pi x : E_1 . E_2 \xleftrightarrow{\mathrm{dB}} \Pi A . M}$$

Fig. 6. De Bruijn and named variables translation

An expression e may be translated into a term, unless some of its free variables are not members of Σ. This specifies the translation function from expressions to terms:

$$\forall (\Sigma, E) .\ (\exists^* t \in \textsc{Trm} .\ \Sigma \vdash E \xleftrightarrow{\mathrm{dB}} t)\ +\ (\exists^* l \neq [] .\ l \subset \mathrm{FV}(E) \backslash \Sigma)$$

The opposite translation is not supposed to fail because it will be used only on terms of the (well-formed) messages generated by the system, and are therefore defined in the current context:

$$\forall (\Sigma, t) .\ \mathrm{AllDiff}(\Sigma)\ \wedge\ |t| < |\Sigma|\ \Rightarrow\ \exists^* E \in \textsc{Expr} .\ \Sigma \vdash E \xleftrightarrow{\mathrm{dB}} t$$

where $|t|$ denotes the highest free de Bruijn variable in term t. This is a bit restrictive: the translation is possible without $\mathrm{AllDiff}(\Sigma)$, as far as t does not refer to hidden variables.

We can prove that it defines a bijection between the domain quotiented by α-conversion and the range:

- two terms equivalent to the same expression are equal:

$$\Sigma \vdash E \overset{\mathrm{dB}}{\longleftrightarrow} t \ \wedge\ \Sigma \vdash E \overset{\mathrm{dB}}{\longleftrightarrow} u \Rightarrow t = u$$

- two expressions equivalent to the same term are α-convertible:

$$\Sigma \vdash E_1 \overset{\mathrm{dB}}{\longleftrightarrow} t \ \wedge\ \Sigma \vdash E_2 \overset{\mathrm{dB}}{\longleftrightarrow} t \Rightarrow E_1 \overset{\alpha}{\equiv} E_2$$

Every structure described in the previous section (commands, messages and errors) is duplicated by changing terms into expressions. The commands are called abstract syntax trees (AST), messages and errors are said printable. The translation predicate is extended to these pairs of sets.

Only the error messages change, because new errors may appear during this translation. Instead of having De Bruijn errors, we may have unbound variable names. The Under constructor also changes: it introduces not only a type expression but also the print name of the variable.

The behavior of the machine is the following: the read AST is processed to build a command. It only consists of translating the expressions into terms. In the future, it could include other syntactic processes (e.g. macro expansions), and this phase would rather be called command synthesis. This operation may fail, and a printable error message is emitted. If it succeeds, the command is given to the internal machine, and the output of the machine (message or error) is translated to make it printable.

The correct behavior is defined by one clause, calling the machine of the previous section:

$$\frac{S_{\mathrm{i}} \vdash a \overset{\mathrm{dB}}{\longleftrightarrow} c \qquad (S_{\mathrm{i}}, c) \overset{\mathrm{exec}}{\longrightarrow} (S_{\mathrm{f}}, m) \qquad S_{\mathrm{i}} \vdash m' \overset{\mathrm{dB}}{\longleftrightarrow} m}{(S_{\mathrm{i}}, a) \overset{\mathrm{step}}{\longrightarrow} (S_{\mathrm{f}}, m')}$$

Note that the message is translated in the initial state S_{i}, because we feel that the output message should be understood without knowing exactly the effect of the command.

The failure behavior needs two rules, because errors may be raised either during synthesis, or during the execution of the command:

$$\frac{l \neq [\,] \qquad l \subset \mathrm{FV}(a) \backslash \Sigma}{((\Gamma, \Sigma), a) \overset{\mathrm{fail}}{\longrightarrow} \mathrm{UnboundVars}(l)}$$

$$\frac{S_{\mathrm{i}} \vdash a \overset{\mathrm{dB}}{\longleftrightarrow} c \qquad (S_{\mathrm{i}}, c) \overset{\mathrm{error}}{\longrightarrow} e \qquad S_{\mathrm{i}} \vdash e' \overset{\mathrm{dB}}{\longleftrightarrow} e}{(S_{\mathrm{i}}, a) \overset{\mathrm{fail}}{\longrightarrow} e'}$$

The machine is now fully specified, and we can prove that any AST in any state may be answered either by a message or an error:

$$\forall S_{\mathrm{i}}, a.\ \exists^{*}(S_{\mathrm{f}}, m).\ (S_{\mathrm{i}}, a) \xrightarrow{\text{step}} (S_{\mathrm{f}}, m)\ +\ \exists^{*} e.\ (S_{\mathrm{i}}, a) \xrightarrow{\text{fail}} e$$

This theorem gives, by extraction, the AST interpreter. The difficulty is to ensure that no de Bruijn translation error occurs in the generated messages. This holds because the members of the messages are well-typed (the constructor ExpectType requires an extra hypothesis for that).

Finally, we check that the steps are not ambiguous:

$$\neg(\ (S_{\mathrm{i}}, a) \xrightarrow{\text{step}} (S_{\mathrm{f}}, m)\ \ \wedge\ \ (S_{\mathrm{i}}, a) \xrightarrow{\text{fail}} e\ )$$

This requires the synthesis step to be deterministic. Or else we could imagine that an AST could be translated into two commands, one succeeding, and the other one failing.

6 Building the System

We performed the extraction of the AST interpreter. The output is an Objective Caml program that takes the current state and an AST and returns either the new state, with an acknowledgment message, or an error message. This function is used to make a single step.

The effective system iterates this function on an input stream of commands and yields a stream of messages. Having acknowledgment and error messages in different types distinguishes two behaviors w.r.t the errors:

- The usual behavior of a compiler would be to abort when an error occurs.
- A toplevel would just print the error message, and resume with the same state, displaying a prompt.

We write a toplevel loop by hand. Since such a loop may potentially not terminate, it must be specified in Coq using co-inductive types, but during the extraction, they cannot be easily mapped to Objective Caml, which is a strict language.

The input stream is not a command stream, but a character one. We have to write a parser. In the same way, the output stream should also be a character stream. We also write a pretty-printer for the messages. Since the concrete syntax is close to the abstract syntax, these functions are very easy to write, and are assumed correct.

These programs may be pasted together and make up a stand-alone system, and the previous sections form its user manual. As an example, we can define the set of naturals and check a few terms:

```
Coc < Axiom nat:Prop.
nat is assumed.
Coc < Axiom O:nat.
```

```
O is assumed.
Coc < Axiom S:nat->nat.
S is assumed.
Coc < Infer (S (S O)).
Inferred type: nat
Coc < Check [x:nat](S (S x)) : nat->nat.
Correct.
```

The examples below show how the system reacts when entering a command that would violate the invariant:

```
Coc < Axiom S:Prop.
Error: Name clash with S.
Coc < Infer (S S).
Error: The term S of type nat->nat cannot be applied
 to S of type nat->nat.
```

These messages are, as expected, relevant and suggest precisely the nature of the error. But there is a problem with local contexts:

```
Coc < Infer [A:Prop][x:A](x x).
Error: In the local context
 x0 : Prop
 x1 : x0
The term x1 of type x0 is non-functional.
```

The message is very difficult to read because the names given by the user are changed. This happens because the internal representation of terms does not handle names. The latter are simply forgotten during the synthesis process. Only the objects introduced at toplevel have a name.

We may consider two ways to fix this problem, by changing the structure of terms. First, adopting a named-based presentation, like in [17, 18]. This makes the metatheory more difficult to prove, because we do not use the usual structural induction scheme, but a scheme taking the α-conversion into account.

We would suggest keeping the de Bruijn notation, and annotating the binding constructions λ and Π with names. The latter are seen as basenames for the names actually printed. These annotations would simply be ignored in all kernel operations (typing, reduction, substitution, etc.). The name capture problem would be dealt with at printing, by adding a suffix to discriminate the binders with the same basename.

Formally, this is not as trivial as it may seem, because we have to reason modulo a pseudo-α-conversion rule, instead of the handy Leibniz equality. The problem may be serious when proving the confluence property. For example, if we consider the η-reduction, the Calculus of Constructions without names is confluent, but the system with names is not (w.r.t. Leibniz equality): $\lambda x.((\lambda y.y)\ x)$ β-reduces to $\lambda x.x$, but η-reduces to $\lambda y.y$. This issue also occurs in the name-based presentation.

7 Conclusion

All the results about the interface are rather easy to prove: the main difficulty is to find a presentation which is both intuitive and concise. The formalization of the translation between de Bruijn and named variables notations is sometimes a bit tedious. This is due to the initial choice of using de Bruijn variables, and then building a thin layer of names on top of it.

The command language we proved is still a very low-level one. The natural extension of this work would be to include more elaborate mathematical concepts, e.g. the section mechanism that initially motivated this approach.

Another direction to explore would be to prove the parser and the pretty-printer. This is not our concern for the moment: one can always want to go further in the formalization. But, if we do, we will have to go on and prove the file system, and then, the operating system, and so on. We would better try to find a limit to what we prove in Coq. Our choice of assuming the parser and the printer to be correct is pragmatic: such functions are easy to write, and tools such as Yacc may help.

Since this development was inspired from the reference manual, we may think that it can in fact become the documentation itself. Of course, it would ask the user to be able to read a Coq script before he reads the Coq documentation, which seems contradictory. But, extrapolating Yann Coscoy's work on the extraction of textual explanations from proof-terms [8], we can imagine that the user manual in natural language could be an extracted version from this formal manual.

Acknowledgments

I would like to thank Benjamin Werner for urging me on writing this article, and I owe him a lot for his precious comments on what error messages should look like. Some aspects that needed more explanation and the references to related works have been improved thanks to the (anonymous) referees.

References

1. Barendregt, H.: Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Vol II, Elsevier, 1992.
2. Barras, B.: Coq en Coq. Rapport de Recherche INRIA 3026. Octobre 1996.
3. Barras, B., Werner, B.: Coq in Coq. *Submitted to publication.* `http://pauillac.inria.fr/~barras/coqincoq.ps.gz`
4. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliâtre, J.-C., Giménez, E., Herbelin, H., Huet, G., Muñoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saïbi, A., Werner, B.: The Coq Proof Assistant Reference Manual Version 6.1. Technical Report 0203. Coq Project-INRIA Rocquencourt-ENS Lyon. May 97.
5. Boyer, R.S., Dowek, G.: Towards Checking Proof-Checkers. In Herman Geuvers, editor, *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs*, May 1993.

6. De Bruijn, N.J.: 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.
7. Coquand, T., Huet, G.: The Calculus of Constructions. In: *Information and Computation Vol. 76, February/March 1988* (ed. A.R.Meyer), Academic Press, London, 95-120.
8. Coscoy, Y., Khan, G., Théry, L.: Extracting text from proofs. INRIA Research Report 2459. January 1995.
9. Courant, J.: A Module Calculus for Pure Type Systems. In: *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications, TLCA'97, Nancy* (ed. Ph. de Groote and J. R. Hindley), Springer-Verlag, LNCS 1210, April 1997.
10. Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. *Cambridge Tracts in Theoretical Computer Science 7*. Cambridge University Press.
11. Geuvers, H., Nederhof, M.-J.: A Modular Proof of Strong Normalization for the Calculus of Constructions. *Journal of Functional Programming*, 1(2):155-189, April 1991.
12. Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF: A Mechanized Logic of Computation. *Springer-Verlag*, LNCS 78, 1979.
13. Huet, G.: The Constructive Engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. WorldScientific Publishing, 1989. Commemorative Volume for Gift Siromoney.
14. Leroy, X.: Applicative Functors and Fully Transparent Higher-Order Modules. In: *22nd Symposium on Principles of Programming Languages*, pp. 142-153. ACM Press, 1995.
15. Martin-Löf, P.: A Theory of Types. Technical Report 71-3, University of Stockholm, 1971.
16. Paulin-Mohring, C., Werner, B.: Synthesis of ML programs in Coq. *Journal of Symbolic Computation–special issue on automated programming*,1993.
17. Pollack, R.: Closure Under Alpha-Conversion. In: *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, `file://ftp.dcs.ed.ac.uk/pub/lego/alpha_closure.ps.gz`, (ed. H. Barendregt and T. Nipkow), Springer-Verlag, pp. 313–332, LNCS 806, 1994.
18. Pollack, R.: A Proof Checker for the Extended Calculus of Constructions. Ph. D. Thesis, `ftp://ftp.dcs.ed.ac.uk/pub/lego/thesis-pollack.ps.Z`, University of Edinburgh, 1994.
19. Pollack, R.: A Verified Typechecker. In: *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications, TLCA'95, Edinburgh*, (ed. M. Dezani-Ciancaglini and G. Plotkin), Springer-Verlag, LNCS 902, April 1995.

This article was processed using the LaTeX macro package with LLNCS style