# Mathematical Libraries as Proof Assistant Environments

Claudio Sacerdoti Coen*

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna, ITALY
sacerdot@cs.unibo.it

**Abstract.** In this paper we analyse the modifications on logical operations — as proof checking, type inference, reduction and convertibility — that are required for the identification of a proof assistant environment with a distributed mathematical library, focusing on proof assistants based on the Curry-Howard isomorphism.
This identification is aimed at the integration of Mathematical Knowledge Management tools with interactive theorem provers: once the distinction between the proof assistant environment and a mathematical library is blurred, it is possible to exploit Mathematical Knowledge Management rendering, indexing and searching services inside an interactive theorem prover, a first step towards effective loosely-coupled collaborative mathematical environments.

## 1 Introduction

The main goal of Mathematical Knowledge Management (MKM) is the creation of large distributed libraries of mathematical knowledge and the development of tools to manage, render, data mine, index and retrieve the notions in these libraries. Mathematical Knowledge Management does not study the process of new mathematical knowledge creation by a mathematician, but is expected to have a substantial impact on it: the better the existent knowledge is grasped and retrieved, the quicker the process of creation. For instance, recent studies [2, 3] implement an authomatic knowledge discovery framework by tightly integrating Mathematical Knowledge Management with authomatic theorem proving: the existent mathematical knowledge is used to derive new definitions, to guess interesting properties and to drive the theorem prover by suggesting well known proof techniques; the theorem prover is used to detect the properties that hold and prune out wrong guesses.

It is not known whether all the expectations will be met or whether the mathematicians will not adopt the Mathematical Knowledge Management tools finding them useless. Nevertheless, we already have evidence of the necessity for

* Partially supported by 'MoWGLI: Math on the Web, Get it by Logic and Interfaces', EU IST-2001-33562

them in the restricted domain of interactive theorem proving. The mathematician that deals with formal mathematics has to develop her own theories on top of large bases of apparently trivial facts that nevertheless need to be proved and retrieved. Mathematical Knowledge Management techniques are necessary since large knowledge bases can be developed only collaboratively and since often the facts in them are so obvious event not to have a name easy to remember, so to be effectively retrieved only using the advanced indexing techniques developed in MKM.

Proof assistants — also called interactive theorem provers — are based on the notion of *environment*, that is the knowledge base that is local to one development and that is interactively augmented. Logical operation (as proof-checking) and extra-logical utilities (as

rendering and searching) operate on the proof assistant environment only.

To achieve full integration with Mathematical Knowledge Management tools, the natural path is blurring the distinction between a distributed mathematical library and the environment of a proof assistant, consequently replacing every extra-logical utility that is limited to the environment only with the corresponding Mathematical Knowledge Management tool that operates over the whole distributed library.

In this paper we analyse the necessary modifications on the logical operations for the identification of the proof assistant environment with a distributed mathematical library, focusing on the proof assistants based on the Curry-Howard isomorphism.

In Sect. 2 we present the Calculus of Constructions as a paradigmatic example of the logic of a proof assistant based on the Curry-Howard isomorphism. In particular, we detail the classical presentation of the reduction and typing rules of the calculus, and we analyse the role of the environment in this presentation.

Since the rules are not syntax oriented, implementations are based on an alternative version that also handles the environment in a completely different way. This operational presentation of the rules is described in Sect. 3. Both set of rules are not well suited for the replacement of environments with distributed mathematical libraries.

In Sect. 4 we present a third original version of the rules that differs from the operational presentation in the way environments are constructed: instead of the operational "bottom up" construction of the environment, based on well-typed extensions of the current environment, we present a "top down" approach where the environment is considered to be a subset of the mathematical library and where the environment necessary to type-check a definition can be dynamically inferred during type-checking.

Intuitively, in the classical approach the environment is a consistent space of mathematical notions, that can only be augmented if its consistency is preserved. No notion outside this consistent space is accessible and no operation can be performed on it. In our approach, instead, there is a much larger space of connected and maybe conflicting mathematical notions, that is the mathematical library. We can associate to every point in the space an environment that is the minimal consistent subspace that contains the point and that is closed for references (i.e. occurrences of lemmas and definitions). It is possible to operate on the whole space as long as inconsistent subspaces are not considered at once.

Section 4 analyses the practical relevance of the contribution presented in this paper, whereas Sect. 5 describes future work.

## 2 The Calculus of Constructions

In this section we present a variant of the Calculus of Constructions (CoC) [5] as a paradigmatic example of a calculus used to implement proof assistants based on the Curry-Howard isomorphism. The given presentation is obtained by removing from the Calculus of (Co)Inductive Constructions — the calculus of the Coq proof assistants[1] — all the rules that deal with local definitions and (co)inductive types. The part of the calculus that deals with universes is preserved, but is simplified (hence making it less expressive). The motivation is the complexity of the omitted rules, that are largely orthogonal to the topic addressed in this paper. Nevertheless, in the author's PhD. dissertation [9] the whole Calculus of (Co)Inductive Constructions is considered.

Since CoC is (an extension of) a Pure Type System, we could adopt Pure Type Systems extended with constants as our paradigmatic example. Nevertheless, the generalization of what we propose to Pure Type Systems is a trivial exercise and casts no additional light on the topic. Moreover, our formalization has the additional benefit of being closer to the calculus actually implemented in the proof assistant Coq.

### 2.1 Syntax

Let $\mathcal{V}$ be a denumerable family of variable names, $x, y, z, \ldots \in \mathcal{V}$.
Let $\mathcal{C}$ be a denumerable family of constant names, $c, c_1, \ldots, c_n \in \mathcal{C}$.
Well formed terms (represented by $t$,$f$,$u$,$T$,$U$,$N$,$M$) are inductively defined as follows.

$$\begin{array}{rll} t ::= & x & \text{identifiers} \\ \mid & c & \text{constants} \\ \mid & \mathbf{Set} \mid \mathbf{Prop} \mid \mathbf{Type}(j) & \text{sorts} \\ \mid & (t\ t) & \text{application} \\ \mid & \lambda x : t.t & \lambda\text{-abstraction} \\ \mid & \Pi x : t.t & \text{dependent product} \end{array}$$

Identifiers are references to locally bound variables, whereas constants are references to global definitions that are collected in an ordered list called an *environment*. The terms **Set**, **Prop** and **Type**$(i)$ are called sorts and are indexed by $s$. Sorts are the types of a type. In particular, the sorts $Prop$ and $Set$ are used to type respectively propositions and data types. Every sort is typed by another sort $Type(i)$ characterized by a greater index $i$ (the index of $Prop$ and

[1] `http://coq.inria.fr`

*Set* is 0). A $\lambda$-abstraction $\lambda x : t_1.t_2$ is a function whose input $x$ is of type $t_1$ and whose output is the expression $t_2$. A dependent product $\Pi x : t_1.t_2$ is the type of all the functions whose input $x$ is of type $t_1$ and whose output is of type $t_2$. The variable $x$ is bound in $t_2$ both in $\lambda$-abstractions and dependent products. Free variables, substitution and $\alpha$-conversion are defined as usual. Terms are syntactically identified up to $\alpha$-conversion.

### 2.2 Environments and Contexts

As environments are ordered lists used to collect global definitions, contexts are ordered lists used to collect local declarations. Contexts and environments are formally defined by the following grammar:

$$\begin{array}{rll} \Gamma ::= & [] & \text{empty context} \\ \mid & \Gamma.(x : T) & \text{declaration of } x \text{ of type } T \\ E ::= & [] & \text{empty environment} \\ \mid & E.(c : T := t) & \text{definition of } c \text{ as } t \text{ of type } T \end{array}$$

We assume that no variable $x$ is declared twice in a context and that no constant $c$ is declared twice in an environment. The typing rules of the calculus enforce this invariant.

We write $(x : T) \in \Gamma$ if $x$ is declared in $\Gamma$ to have type $T$. Similarly, we write $(c : T := t) \in E$ if $c$ is defined in $E$ to be $t$ of type $T$.

### 2.3 Reductions and Convertibility

The syntax of the CoC allows to define and apply functions. The reduction rules of the calculus describe how the result of a function application is computed. We define two reduction rules: $\delta$-reduction and $\beta$-reduction.

$\delta$-reduction states that a constant occurrence can be replaced with its definiendum:

$$c \rhd_\delta t \quad \text{if } (c : T := t) \in E$$

$\beta$-reduction substitutes the application of a function to its argument with the body of the function, replacing the formal parameter with the actual argument:

$$(\lambda x : T.M\ N) \rhd_\beta M\{N/x\}$$

Additionally, the substitution operator $M\{N/x\}$ takes care of avoiding capturing of free variables of $N$.

The reflexive, symmetric, transitive and context-aware closure $=_{\beta\delta}$ of $\delta$-reduction and $\beta$-reduction is called convertibility. The reduction rules form a strongly normalizable rewriting system [11]. As a consequence, convertibility is a decidable property: to decide whether two terms are convertible we can test $\alpha$-convertibility of their normal forms. More efficient algorithms based on weak head reduction are used in real world implementations [8, 9].

Since convertibility is a decidable equivalence relation, it is practical to identify terms up to convertibility. Concretely, this means reducing the number of human provided deduction steps of a proof by replacing them with machine performed computational steps.

### 2.4 Typing Rules

We give the typing rules of CIC by mutually defining two judgements. The first one, $E[\Gamma] \vdash t : T$, asserts that $t$, in an environment $E$ and a context $\Gamma$ that depends on $E$, has type $T$. The second one, $\mathcal{WF}(E)[\Gamma]$, asserts that $E$ is a valid environment — i.e. it is acyclic and it comprises only well-typed definitions — and that $\Gamma$ is a valid context in $E$ — i.e. it is acyclic and it comprises only well-typed variable declarations. An environment $E$ is acyclic when each constant definition $d$ may refer only constants defined before $d$ in $E$. Similarly, a context $\Gamma$ in $E$ is acyclic when each variable declaration $d$ may refer only constants in the environment $E$ or variables declared before $d$ in $\Gamma$. Cyclic environments and contexts must be avoided since in general they correspond to non well-founded (and logically inconsistent) recursive definitions. Finally, we will see that $E[\Gamma] \vdash t : T$ implies $\mathcal{WF}(E)[\Gamma]$ for each couple of terms $t$ and $T$.

The rules of the two judgements are given in Table 1. The Conv rule is the one responsible for the identification of convertible terms. The version proposed here is a major simplification of the corresponding rule adopted in the Coq system and adapted from ECC [7]. In the original rule the equivalence relation $=_{\beta\delta}$ is replaced by an order relation $\leq_{\beta\delta}$ that admits cumulativity between universes. Cumulativity grants that each term of type $\mathbf{Type}(i)$ is also of type $\mathbf{Type}(j)$ for each $j > i$ (and is expressed by the rule $\mathbf{Type}(i) \leq_{\beta\delta} \mathbf{Type}(j)$. The order relation can also be extended to relate non-convertible function spaces (and sigma types, if available) when their input types (in contra-variant position) are convertible and their output types (in co-variant position) are in $\leq_{\beta\delta}$. The choice of the less expressive version of the rule avoids further complications that are orthogonal to our proposal.

This version of the typing rules is the one usually presented in theoretical dissertations about the Calculus of Constructions, since it is the presentation that more easily allows to prove metatheoretical results. Notice, however, that this formulation is not syntax directed: to type-check a term $t$ it is always possible to apply both the Conv rule and the rule — or the rules when $t$ is a dependent product — that match the syntactic structure of the term. Thus implementations are usually based on an alternative syntax directed presentation of the typing rules of the calculus.

In Sect. 3 we will present the syntax directed version of the typing rules of the calculus. Before that, let's analyze the way the environment is handled in the typing rules already given.

### 2.5 Environment Handling

First of all, by inspection of the typing rules of the $E[\Gamma] \vdash t : T$ judgement we notice that the environment $E$ is never modified (i.e. the same environment is used both in the conclusion and in every hypothesis). Moreover, every internal node of the derivation tree but conversion nodes (rules Prod-SP–Conv) does not directly exploit the environment. The conversion rule access the environment to compute $\delta$-reduction steps, but without checking its validity. On the contrary, WF-Empty

$$\mathcal{WF}([])[[]]$$

WF-Var

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathbf{Prop}, \mathbf{Set}, \mathbf{Type}(i)\} \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma, (x : T)]}$$

WF-Const

$$\frac{E[[]] \vdash T : s \qquad s \in \{\mathbf{Prop}, \mathbf{Set}, \mathbf{Type}(i)\} \qquad E[[]] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E; c : T := t)[[]]}$$

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathbf{Prop} : \mathbf{Type}(n)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathbf{Set} : \mathbf{Type}(n)}$$

Ax-Acc

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad n < m}{E[\Gamma] \vdash \mathbf{Type}(n) : \mathbf{Type}(m)}$$

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T := t) \in E}{E[\Gamma] \vdash c : T}$$

Prod-SP

$$\frac{E[\Gamma] \vdash T : s_1 \qquad E[\Gamma, (x : T)] \vdash U : s_2 \qquad s_1 \in \{\mathbf{Prop}, \mathbf{Set}\} \quad \text{or} \quad s_2 \in \{\mathbf{Prop}, \mathbf{Set}\}}{E[\Gamma] \vdash \Pi x : T.U : s_2}$$

Prod-T

$$\frac{E[\Gamma] \vdash T : \mathbf{Type}(n_1) \qquad E[\Gamma, (x : T)] \vdash U : \mathbf{Type}(n_2) \qquad n_1 \leq n \qquad n_2 \leq n}{E[\Gamma] \vdash \Pi x : T.U : \mathbf{Type}(n)}$$

Lam

$$\frac{E[\Gamma, (x : T)] \vdash t : U \qquad E[\Gamma] \vdash \Pi x : T.U : s}{E[\Gamma] \vdash \lambda x : T.t : \Pi x : T.U}$$

App

$$\frac{E[\Gamma] \vdash t : \Pi x : U.T \qquad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{u/x\}}$$

Conv

$$\frac{E[\Gamma] \vdash T_1 : s \qquad E[\Gamma] \vdash t : T_2 \qquad E[\Gamma] \vdash T_1 =_{\beta\delta} T_2}{E[\Gamma] \vdash t : T_1}$$

**Table 1.** Typing rules, original formulation.

all the leaves of a derivation tree (rules Ax-Prop–Const) check the validity of the environment by testing $\mathcal{WF}(E)[\Gamma]$. Due to our previous observation, this means that a proof of $\mathcal{WF}(E)[\Gamma]$ for the same version of $E$ and $\Gamma$ is required at each leaf of the derivation tree. Of course, this is totally unreasonable from an implementation point of view. Thus the operational presentation of the rules will take care of this, granting that a couple $(E, \Gamma)$ will be checked only once.

Let's focus now on the derivation rules of the $\mathcal{WF}(E)[\Gamma]$ judgement. The applications of this rules are stratified: first of all the WF-Var rule is applied several times to remove one after the other all the declarations in the context $\Gamma$; then the WF-Const rule is applied several times to remove one after the other all the definitions in the environment $E$. Finally a single application of the WF-Empty rule ends the proof. In the first phase — the applications of the WF-Var rule — the environment $E$ is also never modified.

From the previous observations we conclude that the only rule that destructures the environment $E$ is the WF-Const rule. The WF-Const rule checks for well-formedness of the environment $E'$ obtained removing from $E$ its last element; then it checks for the well-typedness of the definition in the environment $E'$. By induction on the height of the derivations we can easily prove that eventually every element of $E$ is checked for well-formedness in a derivation of $\mathcal{WF}(E)[]$. Interestingly, however, this check is apparently done "on demand": the well-formedness of the environment $E$ is not checked *before* starting the proof of $E[\Gamma] \vdash t : T$, but only during its proof, whenever an occurrence of a constant is met. However, every constant in $E$ is checked, even if it does not occur in $\Gamma$, $t$ and $T$. In the latter case, the thinning lemma [11] states the uselessness of the check, since unused constants can be removed from the environment without affecting well typedness. This suggests that checking "on demand" can be improved to avoid unused definitions.

The previous observations imply that the standard presentation of the type-checking rules is not appropriate for a direct implementation, especially if the environment is substituted by a large and distributed mathematical library, since

- the parts of the library that are not useful to type-check the term under consideration are checked anyway;
- each object in the library is checked more than once.

In the next section we analyse the standard operational presentation of the type-checking rules, discussing how the environment is handled.

## 3 Syntax Directed Type Checking Rules

The syntax directed (or operational) version of the reduction and type-checking rules of CoC solves at once two problems of the previous presentation: each definition in the environment is checked at most once; and at most one type-checking or reduction rule can be applied to a given term (respectively to a couple of terms).

Thanks to the latter remarkable property the operational presentation of the rules yields an efficient algorithm that does not require backtracking. However, in this paper we are more interested in the first property.

The key idea that underlies the management of the environment is the following already noticed invariant: the environment is consulted, but it is never modified during the type-checking of a term $t$ in the context $\Gamma$. Thus it is possible to define the (current) environment as an abstract data type with two methods: the first one returns the definition of a constant $c$ in the environment; the second one appends a definition to the environment. The latter method checks for the well-typedness of the definition before appending it. The initial environment is the empty environment (that is always well-formed). The invariant that is granted by the two method implementations is the well-formedness of the environment. Thus, in the following typing rules, we can always assume the environment $E$ to be well-typed.

The expression $whd(E, t)$ that occurs in the typing rules compute the weak head normal form of $t$ in the environment $E$. The judgement $E[\Gamma] \vdash t \downarrow t'$ is a syntax directed formulation of the convertibility judgement. It satisfies the property $E[\Gamma] \vdash t \downarrow t'$ iff $E[\Gamma] \vdash t =_{\beta\delta} t'$. A set of rules for the $E[\Gamma] \vdash t \downarrow t'$ judgement that yield and efficient implementation can be found in [8, 9].

The rules of the typing judgement are given in Table 2. The premise $(c : T := t) \in E$ of the BU-Const rule is implemented by calling the first method of the abstract data type that implements $E$ on the constant name $c$. The definition $(c : T := t)$ is the result of the method invocation.

To grant the well-formedness of the environment, a definition $(c : T := t)$ is appended by the second method only if the following conditions are satisfied:

$$c \notin E \quad E[[]] \vdash T : S \quad whd(E[[], T) = s \quad E[[]] \vdash t : T' \quad E[[]] \vdash T' \downarrow T \qquad (*)$$

As already stated in the introduction of the paper, the major drawback of this management of the environment is that the environment must be constructed in a “bottom up” fashion, checking in advance all the results that will be needed for the mathematical result the user wants to develop. Since it is common to discover the need for a lemma during the development of a proof, the bottom up environment construction can be an annoying limitation.

For instance, let us consider the following scenario: the user knows (or finds using a search engine) that there is a lemma in the mathematical library that she would like to use. The lemma is not part of the current environment and it depends on other lemmas and definitions that are not part of the current environment. Thus she is obliged to add to the environment all these lemmas and definition *in the right order*, i.e. without appending a definition that relies on another definition not appended yet. If she is able to append all the lemmas and definitions, comprising the wanted lemma, then the obtained environment is well-formed and she can apply the lemma.

Since we cannot expect the user to perform by hand the previous mechanical operations, we can delegate them to an external software agent. However, this solution is not efficient, since all the terms of all the definitions that must be

BU-Ax-Prop

$$\frac{}{E[\Gamma] \vdash \mathbf{Prop} : \mathbf{Type}(i)}$$

BU-Ax-Set

$$\frac{}{E[\Gamma] \vdash \mathbf{Set} : \mathbf{Type}(i)}$$

BU-Ax-Acc

$$\frac{n \leq m}{E[\Gamma] \vdash \mathbf{Type}(n) : \mathbf{Type}(m)}$$

BU-Var

$$\frac{(x : T) \in \Gamma}{E[\Gamma] \vdash x : T}$$

BU-Const

$$\frac{(c : T := t) \in E}{E[\Gamma] \vdash c : T}$$

BU-Prod

$$\frac{\begin{array}{ll} E[\Gamma] \vdash T : S_1 & whd(E[\Gamma], S_1) = s_1 \\ E[\Gamma, (x : T)] \vdash U : S_2 & whd(E[\Gamma, (x : T)], S_2) = s_2 \end{array}}{E[\Gamma] \vdash \Pi x : T.U : s_2}$$

BU-Lam

$$\frac{E[\Gamma] \vdash T : S \qquad whd(E[\Gamma], S) = s \qquad E[\Gamma, (x : T)] \vdash t : U}{E[\Gamma] \vdash \lambda x : T.t : \Pi x : T.U}$$

BU-App

$$\frac{E[\Gamma] \vdash t : S \qquad whd(E[\Gamma], S) = \Pi x : U.T \qquad E[\Gamma] \vdash u : U' \qquad E[\Gamma] \vdash U \downarrow U'}{E[\Gamma] \vdash (t\ u) : T\{u/x\}}$$

**Table 2.** Typing rules, operational formulation.

added to the environment are browsed twice: the first time to collect all their dependencies to compute their topological sorting; the second time to type-check them. Moreover, bugs in the topological sorting algorithm are difficult to detect since they manifest themselves by producing non well-formed environments whereas well-formed environments can be produced.

Finally, notice that the syntax directed type-checking judgement is not a complete formal description of the type-checker, since the data type and its implementation must also be described, as well as the additional constraints (*).

In the following section we propose yet an alternative formulation of the type-checking judgement. The formulation proposed is syntax oriented, yielding an algorithm that can be mechanically implemented. The environment is managed in a "top down" fashion, being constructed (identified) only on demand, in a "call-by-need" fashion. Concretely, an environment is a consistent subset of a larger (and possibly inconsistent) mathematical library. Thus the user is always free to use at any time a lemma in the library that is not yet in the environment. Finally the judgement proposed is self-contained, no longer requiring an external data type.

## 4 Mathematical Libraries as Environments

Let $\mathcal{L}$ be a (possibly inconsistent) mathematical library (i.e. a set of constant definitions $(c : T := t)$). An environment $E$ is a consistent subset of $\mathcal{L}$. We define two mutual judgements, $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ and $\mathit{fetch}(\Phi, E, c, E', (c : T := t))$, with the following semantics:

- $\mathit{fetch}(\Phi, E, c, E', (c : T := t))$ holds when the environment $E \subseteq \mathcal{L}$, assumed to be consistent, can be extended to a new consistent environment $E' \subseteq \mathcal{L}$ such that $E$ is a prefix of $E'$, no constant in the set of constant names $\Phi$ is defined in $E'$ and $(c : T := t) \in E'$ where $(c : T := t) \in \mathcal{L}$.
  Concretely, the judgement defines a function whose input is the triple $(\Phi, E, c)$ and whose output is the couple $(E', c : T := t)$. We say that $E'$ is the *(minimal) context extension* of $E$ that includes $c$. The set of constant names $\Phi$ is a technical trick used to detect cycles in the constructed environment extension.
- $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ holds when $t$ has type $T$ in the environment $E'$ and in the context $\Gamma$ and $E'$ is an extension of $E$ (assumed to be consistent) such that all the constants in the set of constant names $\Phi$ are not defined in $E'$.
  Concretely, the judgement defines a function whose input is the tuple $(\Phi, E, \Gamma, t, T)$ and whose output is the minimal environment extension $E'$ in which the input is well-typed.

The rules of the two judgements are given in Table 3. The following two theorems, whose proof can be found in the author's PhD. dissertation [9], state the equivalence of the proposed judgement with the standard presentation.

**Theorem 1 (Soundness).** *For each environment $E$, context $\Gamma$, list of references $\Phi$ and term $t$ such that the intersection between the domain of $E$ and $\Phi$ is empty, if $\mathcal{WF}(E)[\Gamma]$ and $\langle\Phi\rangle E[\Gamma] \vdash t : T \bowtie E'$ then*

- $\mathcal{WF}(E')[\Gamma]$
- *the intersection between the domain of $E'$ and $\Phi$ is empty*
- *for each couple of terms $t'$, $T'$ such that $E[\Gamma] \vdash t' : T'$ we also have $E'[\Gamma] \vdash t' : T'$*
- $E'[\Gamma] \vdash t : T$.

**Theorem 2 (Completeness).** *For each environment $E$, context $\Gamma$ and term $t$, if $\mathcal{WF}(E)[\Gamma]$ and $E[\Gamma] \vdash t : T$, then there exists an unique term $T'$ such that $\langle\emptyset\rangle E[\Gamma] \vdash t : T' \bowtie E$ and $E[\Gamma] \vdash T =_{\beta\delta} T'$.*

### Practical Relevance

The main aim of this paper is the development of a type-checker that blurs as much as possible the distinction between the logical environment and the mathematical library, giving to the user the feeling of building her own development on top of the whole library. All the mathematical results in the library must

TD-Ax-Prop

$$\overline{\langle\Phi\rangle E[\Gamma] \vdash \mathbf{Prop} : \mathbf{Type} \bowtie E}$$

TD-Ax-Set

$$\overline{\langle\Phi\rangle E[\Gamma] \vdash \mathbf{Set} : \mathbf{Type} \bowtie E}$$

TD-Ax-Acc

$$\overline{\langle\Phi\rangle E[\Gamma] \vdash \mathbf{Type} : \mathbf{Type} \bowtie E}$$

TD-Var

$$\frac{(x:T) \in \Gamma \quad or \quad (x:T:=t) \in \Gamma}{\langle\Phi\rangle E[\Gamma] \vdash x : T \bowtie E}$$

TD-Const

$$\frac{fetch(\Phi, E, c, E', (c:T:=t)) \qquad \langle\Phi\rangle E'[\Gamma]}{\langle\Phi\rangle E[\Gamma] \vdash c : T \bowtie E'}$$

TD-Prod

$$\frac{\begin{array}{ll}\langle\Phi\rangle E[\Gamma] \vdash T : S_1 \bowtie E' & whd(E'[\Gamma], S_1) = s_1 \\ \langle\Phi\rangle E'[\Gamma, (x:T)] \vdash U : S_2 \bowtie E'' & whd(E''[\Gamma, (x:T)], S_2) = s_2\end{array}}{\langle\Phi\rangle E[\Gamma] \vdash \Pi x : T.U : s_2 \bowtie E''}$$

TD-Lam

$$\frac{\langle\Phi\rangle E[\Gamma] \vdash T : S \bowtie E' \quad whd(E'[\Gamma], S) = s \quad \langle\Phi\rangle E'[\Gamma, (x:T)] \vdash t : U \bowtie E''}{\langle\Phi\rangle E[\Gamma] \vdash \lambda x : T.t : \Pi x : T.U \bowtie E''}$$

TD-App

$$\frac{\langle\Phi\rangle E[\Gamma] \vdash t : S \bowtie E' \quad whd(E'[\Gamma], S) = \Pi x : U.T \quad \langle\Phi\rangle E'[\Gamma] \vdash u : U' \bowtie E'' \quad E''[\Gamma] \vdash U \downarrow U'}{\langle\Phi\rangle E[\Gamma] \vdash (t\ u) : T\{u/x\} \bowtie E''}$$

TD-fetch-Const-hit

$$\frac{(c:T:=t) \in E}{fetch(\Phi, E, c, E, (c:T:=t))}$$

TD-fetch-Const-miss

$$\frac{\begin{array}{ll}c \notin \Phi & (c:T:=t) \in \mathcal{L} \\ \langle\Phi; c\rangle E[[]] \vdash T : S \bowtie E' & whd(E'[[]], T) = s \\ \langle\Phi; c\rangle E'[[]] \vdash t : T' \bowtie E'' & E''[[]] \vdash T' \downarrow T\end{array}}{fetch(\Phi, E, c, E''; (c:T:=t), (c:T:=t))}$$

**Table 3.** Typing rules, proposed new operational formulation.

seem to be always available, even if they have not been type-checked and put in the environment so far.

The typing judgement proposed satisfies at once several properties that altogether yield almost mechanically the implementation of a type-checker that satisfies our needs. Indeed

- the type-checking rules are syntax oriented;
- every object in the environment is checked only once;
- every object in the library is checked at most once, and only if necessary;
- the environment necessary to type-check a term is reconstructed on-the-fly during type-checking;
- the typing judgement is self-contained and fully describes the type-checker kernel.

Soundness (theorem 1) grants two useful implementation properties. The first one is what we name *minimal environment reconstruction*: given a term $t$, if $\langle\emptyset\rangle\emptyset[[]] \vdash t : T \bowtie E$, then $E$ is the smallest well-formed environment required to type-check $t$.

We call the second property *environment extension*: in order to check a new term, definition or declaration for well-formedness, we do not have to throw away the already computed and well-formed environment. We can simply invoke the type-checking procedure passing the current environment as the initial environment. In case of success, the returned environment is the minimal extension of the initial environment that grants the well-formedness property.

Environment reconstruction paves the way to the notion of *trusted information source*. A trusted information source is a subset of the distributed library of knowledge that is closed (in the sense that it is well-formed) and that we can trust to be also consistent. Concretely, trusting an information source $\mathcal{L}'$ amounts to adding a new inference rule that deals with the *fetch* relation, and to slightly change the reduction rule that deals with $\delta$ reduction:
O-fetch-Const-trust

$$\frac{(c : T := t) \in \mathcal{L}'}{\mathit{fetch}(\Phi, E, v, E; (c : T := t), (c : T := t))}$$

The rule for $\delta$–reduction is changed by replacing the premise $(c : T := t) \in E$ with the premise $(c : T := t) \in E$ *or* $(c : T := t) \in \mathcal{L}'$.

Basically, the new rule above means that a declaration or definition in a trusted information source $\mathcal{L}'$ can always be considered as well-formed. Equivalently, it says that $\mathcal{L}'$ can always be considered as a subset of any well-formed environment.

The proposed typing judgement, extended with trusted information sources, has been implemented in a proof-assistant kernel prototype developed in project HELM[2] (Hypertexual Electronic Library of Mathematics) [1] and it has already been in use for a couple of years. The prototype consults the HELM mathematical library of about 40000 lemmas and definitions extracted from the Coq proof

[2] `http://helm.cs.unibo.it`

assistant [10]. The library is physically distributed among several repositories. One of the repositories provides the standard library of the Coq proof assistant, that is granted to be closed and consistent. Thus the repository is considered a trusted information source, whereas all the other repositories are untrusted.

Recently, the kernel of the proof-assistant prototype has been embedded in a fully fledged tactic-based interactive theorem prover, that implements also refinement and unification. Unification is an extension of convertibility that handles missing or incomplete information: two open terms can be unified if there exists a substitution that makes them convertible. In a similar way refinement is an extension of convertibility that handles missing or incomplete information: an open term is refinable when there exists a substitution that makes it well-typed.

In the author's PhD. dissertation [9] the technique presented in this paper is extended to cover also unification and refinement. Thus the user of the HELM proof-assistant is never exposed to the notion of environment: all she knows about is the mathematical library that she can search, render and data mine and that she can freely extend, as long as no inconsistency is added. The searching, rendering and data-mining components of the prototype are off-the-shelf MKM services developed in project HELM and in project MoWGLI[3] (Math on the Web: Get it by Logics and Interfaces).

## 5 Future Work

The Calculus of Constructions and its extensions — as the Calculus of (Co)Inductive Constructions adopted by Coq — is formalized by imposing a linear ordering over the sorts **Type**: $\mathbf{Type}(i) \leq \mathbf{Type}(j)$ whenever $i < j$. The ordering is just used to avoid circularities, that may correspond to logical inconsistencies. No assumption is made in the meta-theory of the calculus about its linearity.

However, implementations of the Calculus of Constructions are usually based on a different sorts representation, where the order relation is a partial order that is automatically inferred by the system and that can be represented as an acyclic graph. The benefits of this solution are twofold: on the one hand the user is given the illusion to work in a simpler (but inconsistent, [4]) impredicative system that has only one sort $\mathbf{Type} : \mathbf{Type}$; on the other hand the system is more flexible in the definition of new sorts and new constraints about the sorts.

For instance, in the classical presentation it is not possible to add a new sort $\mathbf{Type}(i)$ between $\mathbf{Type}(1)$ and $\mathbf{Type}(2)$ unless every sort $\mathbf{Type}(j)$ such that $j \geq 2$ is shifted. Even worse, given two logically unrelated sorts $\mathbf{Type}(n)$ and $\mathbf{Type}(m)$ we are obliged to artificially fix an order — say $n < m$ — unless we are using a partial order. Once the order $n < m$ is fixed, the usage of a lemma that generates the logical constraint $m < n$ is rejected even if the detected cycle does not correspond to a real logical inconsistency.

As a future work, we are planning to detail and implement a version of the typing judgement that is based on a partial order implemented as an acyclic

[3] `http://mowgli.cs.unibo.it`

graph. Since constraints (arcs) among the sorts are introduced during type-checking, we must extend our judgement to build the sorts graph on-demand during environment reconstruction and extension.

One interesting property of this approach with respect to the usual implementation is that the sorts graph inferred by the environment reconstruction procedure is minimal, in the sense that no constraint can be relaxed without losing the well-typedness of the declaration.

Unfortunately, in case of failure during an environment extension we are not allowed to conclude that the given term, declaration or definition is not well-typed. For instance, it may be the case that no well-formed extension of the current environment grants well-formedness, simply because of a conflicting set of constraints. Nevertheless, it may be the case that some of the conflicting constraints are derived from parts of the initial environment that are not necessary to type-check the given input [6]. Thus, to conclude that the input is not well-formed, we are obliged to repeat the check starting from the empty environment.

Notice that the problem we have just noticed is not a side-effect of our special handling of the environment. Indeed, our approach is the only one that allows to overcome the problem, by reconstructing the minimal necessary environment. In the Coq system, for instance, when such a case is met the only solution is to factorize out of the required modules[4] all the definitions necessary to type the incriminated definition. Basically this amounts to perform by hand the operation of environment reconstruction.

It should be noted, however, that the cases where the problem arises are very uncommon. All the cases known by the author are to be found in the encoding of set theories and the theory of categories where big and small sets (or categories) are mixed together in the same development. Nevertheless, once again we feel that, in the context of mathematical knowledge management, the problem must be faced in a serious way, since it can deeply hinder the possibility of extending or reusing a development in the long term.

To integrate the inference of the sorts graph $\gamma$ in the environment reconstruction procedure, all the rules need to be given the sorts graphs an additional input and output. Moreover, the *fetch* relation becomes also responsible for extending the system of constraints by adding the constraints that are required to type-check the reference object. Thus, in order to obtain a sound implementation, we need to keep, for any object in a trusted information source, the minimal set of constraints imposed during its type-checking. Let $\mathcal{K}$ be the map that associates the constraints to a reference $r$. The fetching rule is changed in the following way:

TD-fetch-Const-trust

$$\frac{(c : T := t) \in \mathcal{L}' \qquad \mathcal{WF}(\gamma \sqcup \mathcal{K}(c))}{\mathit{fetch}(\Phi, \langle E, \gamma\rangle, c, \langle E; (c : T := t), \gamma \sqcup \mathcal{K}(c)\rangle, (c(v_1, \ldots, v_n) : T := t))}$$

[4] A Coq module roughly corresponds to a mathematical development. They are the minimal compilation units of the Coq system.

where $\gamma$ is the set of constraints in input, $\sqcup$ is the operation that merges two set of constraints and $\mathcal{WF}(\gamma \sqcup \gamma')$ is the new judgement that checks if the merge of two set of satisfiable constraints is satisfiable (i.e. it does not contain any cycles).

## References


1. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen and I. Schena. "Mathematical Knowledge Management in HELM". In Annals of Mathematics and Artificial Intelligence, 38(1): 27–46, May 2003.
2. B. Buchberger. "Computer-Supported Mathematical Theory Exploration: Schemes, Failing Proof Analysis, and Metaprogramming." Technical report, September 2004, RISC, Austria.
3. S. Colton. Automated Theory Formation in Pure Mathematics Series: Distinguished Dissertations. 2002, XVI, 380 p., ISBN: 1-85233-609-9.
4. T. Coquand. "An Analysis of Girard's Paradox". In Proc. Symposium on Logic in Computer Science, pages 227-236, Cambridge, Massachusetts, 16-18 June 1986. IEEE Computer Society.
5. T. Coquand and G. Huet. "The Calculus of Constructions". Technical report 530, INRIA, May 1986.
6. Judicaël Courant. Explicit universes for the calculus of constructions. In Victor A. Carreño, Cèsar Muñoz, and Sofiène Tahar, editors, Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002, volume 2410 of Lecture Notes in Computer Science, pages 115-130, Hampton, VA, USA, August 2002. Springer-Verlag.
7. Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.
8. R. Harper and R. Pollack. "Type checking with universes". Theoretical Computer Science, 89:107-136, 1991.
9. C. Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving.* PhD. thesis, University of Bologna, 2004.
10. C. Sacerdoti Coen. "From Proof-Assistans to Distributed Libraries of Mathematics: Tips and Pitfalls". In Proc. Mathematical Knowledge Management 2003, Lecture Notes in Computer Science, Vol. 2594, pp. 30–44, Springer-Verlag.
11. B. Werner. *Une Theorie des Constructions Inductives.* PhD. thesis, Université Paris VII, 1994.