Parametricity in an Impredicative Sort

Chantal Keller[1] and Marc Lasson[2]

1 **INRIA Saclay–Île-de-France at École Polytechnique**
`Chantal.Keller@inria.fr`

2 **École Normale Supérieure de Lyon, Université de Lyon, LIP** *
`marc.lasson@ens-lyon.org`

Abstract

Reynold's abstraction theorem is now a well-established result for a large class of type systems. We propose here a definition of relational parametricity and a proof of the abstraction theorem in the Calculus of Inductive Constructions (CIC), the underlying formal language of Coq, in which parametricity relations' codomain is the impredicative sort of propositions. To proceed, we need to refine this calculus by splitting the sort hierarchy to separate informative terms from non-informative terms. This refinement is very close to CIC, but with the property that typing judgments can distinguish informative terms. Among many applications, this natural encoding of parametricity inside CIC serves both theoretical purposes (proving the independence of propositions with respect to the logical system) as well as practical aspirations (proving properties of finite algebraic structures). We finally discuss how we can simply build, on top of our calculus, a new reflexive Coq tactic that constructs proof terms by parametricity.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Calculus of Inductive Constructions; parametricity; impredicativity; Coq; universes.

Digital Object Identifier 10.4230/LIPIcs.CSL.2012.399

1 Introduction

The Coq system [24] is a proof assistant based on the Curry-Howard correspondence: propositions are represented as types and their proofs are their inhabitants. The underlying type system is called the Calculus of Inductive Constructions (CIC in short). In this type system, types and their inhabitants are expressions built from the same grammar and every well-formed expression has a type.

One specificity of Coq among other interactive theorem provers based on Type Theory is the presence of an impredicative sort to represent the set of propositions: `Prop`. Impredicativity means that propositions may be built by quantification over objects which types inhabit any sort, including the sort of propositions (for instance the Agda language has a similar type system except that propositions live in predicative universes [18]). This sort plays a decisive role in the Coq system: in addition to guaranteeing the compositionality of the propositional world, it contains the non-computational content, i.e., expressions meant to be erased by the program extraction process. In particular, it allows the user to add axioms (like the law of excluded middle, axiom of choice, proof irrelevance, etc...) without jeopardizing program extraction.

* UMR 5668 CNRS ENS Lyon UCBL INRIA

Computer Science Logic 2012 (CSL'12).
Editors: Patrick Cégielski, Arnaud Durand; pp. 399–413
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The other sorts are a predicative hierarchy of universes called $\texttt{Type}_0, \texttt{Type}_1, \ldots$. Contrary to Prop, it is stratified: one is not allowed to form a type of a given universe by quantifying over objects of types of higher universes (stratification has been introduced in order to overcome Girard's paradox, see [7] for details). The sort $\texttt{Type}_0$ (also called Set) contains data-types and basic informative types. And $\texttt{Type}_1$ contains types that are quantified over elements of $\texttt{Type}_0$, and so on.

One major component of Coq is its extraction mechanism [15], which produces an untyped term of a ML-like language from any well typed term of Coq. One obvious interest is to obtain certified ML code. Roughly speaking, it proceeds by replacing type annotations and propositional subterms by a dummy constant. A difficulty of program extraction is to decide which terms are informative and which may be erased. The presence of the sort Prop only partially solves this problem in Coq since the system has to distinguish computations over data types from computations over types, although they all live in Type.

In this paper, we propose a new calculus which refines the Calculus of Inductive Constructions, called $\mathsf{CIC_r}$. By adding a new predicative hierarchy of sorts $\texttt{Set}_0$, $\texttt{Set}_1$, ..., it confines the types of all informative expressions and purges the hierarchy $\texttt{Type}_0$, $\texttt{Type}_1$, ... of all computational content. In other words, it guarantees that inhabitants of types in Set are the only expressions which do not disappear during the extraction process.

In spite of that, this new calculus may be naturally embedded into CIC by a very simple forgetful operation. Moreover it remains very close to CIC and in practice only few terms are not representable in $\mathsf{CIC_r}$. That is why it represents a big step towards an implementation in the Coq system.

Being able to identify expressions with computational content – or in other words *programs* – was essential to achieve our initial goal: formalizing the parametricity theory for the Calculus of Inductive Constructions.

Parametricity is a concept introduced by Reynolds [22] to study the type abstraction of System F. It expresses the fact that well-typed programs must behave uniformly over their arguments with an abstract type: if the type is abstract, then the functions do not have access to its implementation. Wadler [26] explained how this could be used to deduce interesting properties shared by all programs of the same type. Later, Plotkin and Abadi [21] introduced a logic in which these uniformity properties can be expressed and proved. This logic may be generalized into a second-order logic with higher-order individuals [23, 27].

The main tools of parametricity theory are logical relations defined inductively over the structure of types together with the so-called *abstraction theorem*, which builds a proof that any closed program is related to itself for the relation induced by its type. For instance the relation induced by the type $\forall\alpha, (\alpha \to \alpha) \to \alpha \to \alpha$ of Church numerals is given by the following definition (represented here in Coq using the standard encoding of relations):

$$\begin{aligned}\lambda(f\,f' : \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha).\forall\alpha\,\alpha'\,(R : \alpha \to \alpha' \to \texttt{Prop})\,(g : \alpha \to \alpha)(g' : \alpha' \to \alpha').\\ (\forall x\,x'.R\,x\,x' \to R\,(g\,x)\,(g'\,x')) \to \forall z\,z'.R\,z\,z' \to R\,(f\,\alpha\,g\,z)\,(f'\,\alpha'\,g'\,z')\end{aligned}$$

The abstraction theorem tells that any closed term F of type $\forall\alpha, (\alpha \to \alpha) \to \alpha \to \alpha$ is related to itself according to this relation.

Recently, the work from Bernardy *et al.* [5] generalized these constructions up to a large class of Pure Type Systems and showed that parametricity theory accommodates well with dependent types. But this cannot be straightforwardly adapted to CIC, because parametricity relations live in higher universes instead of using the standard encoding of relations in Prop. Besides, it is difficult to make parametricity relations live inside Prop while conserving abstraction.

But parametricity in a system like Coq would be profitable: it could lead to more automation, for instance for developing mathematical theories: we give here an example in finite group theory (Section 5.3). Basing on our refined calculus, we started an implementation of a Coq tactic that can build closed instances of the abstraction theorem [1].

The paper is organized as follows. After explaining in details why we need to refine the Calculus of Inductive Constructions, we present CIC_r in Sections 2 and 3. Section 4 is devoted to the definition of relational parametricity, and the proof of the abstraction theorem, without and with inductive types. In Section 5, we present different kinds of "theorems for free" that are derived from the general abstraction theorem, like independence of the law of excluded middle with respect to CIC_r or standard properties of finite groups. We finally explain the algorithm behind the implementation of the Coq tactic (Section 6) before discussing related works and concluding.

2 CIC_r: a refined calculus of constructions with universes

2.1 The need for a refinement

In older versions of CIC, Set was not a synonym for $\mathtt{Type}_0$ but a special impredicative sort containing data-types and basic informative types. However, there is a smaller demand from the users for the impredicativity of Set rather than the possibility to add classical axioms to CIC, and having both may lead to the inconsistency of the system (the conjunction of excluded middle and description conflicts with the impredicativity of Set [10]). As a result, nowadays Set is predicative and behaves in CIC as the first level of the hierarchy of universes.

In CIC_r, we want to reintroduce the sort Set of informative types in order to mark the distinction between expressions with computational content and expressions which are erased during the extraction process. To stay close to CIC, we want Set to be predicative, so we introduce a hierarchy of sorts $\mathtt{Set}_0$, $\mathtt{Set}_1$, ...

In the refinement, the CIC hierarchy of sorts Type is thus divided into two classes : a hierarchy of sorts Set, whose inhabitants have a computational content, and a hierarchy of sorts Type, whose inhabitants are uninformative. There is a difference of level between inhabitants of Set and inhabitants of Type: the inhabitants of Set are inhabited only by non-habitable expressions whereas Type contains the signatures of predicates and type constructors which are themselves, when fully applied, inhabited respectively by proofs and programs.

In Coq, deciding to which of this two classes an expression of type Type belongs is essential in the extraction mechanism. In the original two-sorted calculus of constructions (i.e. without universes), the top-sort contains only arities and therefore the level of terms can be easily obtained by looking at the type derivation and the extraction procedure is simple [19]. However, in Coq, to extract the computational content of an inhabitant of sort Type, the extraction algorithm decides if a type is informative by inspecting the shape of its normal form [16,17]. Therefore termination of extraction relies on the normalization of CIC. It makes the correction of the extraction difficult to formally certify.

2.2 Presentation of the calculus

The syntax of CIC_r is the same as the standard calculus of constructions except that we extend the set of sorts. Terms are generated by the following grammar:

$$A, B \quad := \quad x \;\mid\; s \;\mid\; \forall x : A.B \;\mid\; \lambda x : A.B \;\mid\; (A\,B)$$

$$i < j \;\; \frac{}{\texttt{Set}_i <: \texttt{Set}_j} \; (\textsc{Sub}_{1-1}) \qquad i < j \;\; \frac{}{\texttt{Type}_i <: \texttt{Type}_j} \; (\textsc{Sub}_{1-2})$$

$$\frac{A <: B}{\forall x : C.A <: \forall x : C.B} \; (\textsc{Sub}_2)$$

Figure 1 Subtyping rules

where s ranges over the set $\{\texttt{Prop}\} \cup \{\texttt{Set}_i, \texttt{Type}_{i+1} | i \in \mathbb{N}\}$ of *sorts* and x ranges over the set of *variables*. In the remaining of the paper, when no confusion is possible, $\texttt{Set}$ stands for "$\texttt{Set}_i$ for some i", and $\texttt{Type}$ stands for "$\texttt{Type}_i$ for some i". The notation $i \vee j$ represents the maximum of i and j.

As usual, we will consider terms up to α-conversion and we denote by $A[B/x]$ the term built by substituting the term B to each free occurrence of x in A. The β-reduction $\triangleright$ is defined as in CIC, and we write $A \equiv B$ to denote the β-conversion.

A context Γ is a list of couples $x : A$ where x is a variable and A is term. The empty context is written $\langle\rangle$. The system has subtyping, given by the rules in **Figure 1**. The typing rules of $\mathsf{CIC_r}$ are given in **Figure 2**.

The word *type* is a synonymous for a term that can be typed by a sort following those rules. We call *informative types* inhabitants of $\texttt{Set}$, *programs* inhabitants of informative types, *propositions* inhabitants of $\texttt{Prop}$ and *proofs* inhabitants of propositions. The sort $\texttt{Type}_i$ adds a shallow level to the system; it is populated with two kinds of terms: *arities*, which are terms whose head normal forms have the form $\forall(x_1 : A_1) \dots (x_n : A_n).s$ where s is either $\texttt{Prop}$, $\texttt{Set}_j$ or $\texttt{Type}_j$ with $j < i$; and higher-order functions that manipulate arities, and whose types are arities with $\texttt{Type}_{i+1}$ as a conclusion. We say that a term has some sort s if s is the type of its type.

3 Inductive types

The calculus is extended with inductive definitions and fixpoints. The presentation is very similar to Chapter 4.5 of the Reference Manual of $\texttt{Coq}$ [24], and one can report to it to have further details.

3.1 Inductive types and fixpoints

The grammar of $\mathsf{CIC_r}$ terms is extended with:

$$A, B, P, Q, F \cdots := \cdots \quad | \quad I \quad | \quad c \quad | \quad \texttt{case}_I(A, \vec{Q}, P, \vec{F}) \quad | \quad \texttt{fix}\,(x : A).B$$

We write $\texttt{Ind}^p(I : A, c_1 : C_1, \dots, c_k : C_k)$ to state that I is a well-formed inductive definition typed with p parameters, of arity A, with k constructors $c_1, \dots, c_k$ of respective types $C_1, \dots, C_k$. It requires that:

1. the names I and c_j are fresh;
2. A is a well-typed arity of conclusion $\texttt{Prop}$ of $\texttt{Set}$: it is convertible to $\forall\overrightarrow{(x : P)}^p \overrightarrow{(y : B)}^n.r$ where $r \in \{\texttt{Prop}, \texttt{Set}\}$;
3. for any j, C_j has the form $\forall\overrightarrow{(x : P)}^p \overrightarrow{(z : E_j)}^{n_j}.I\, \vec{x}^p\, \vec{D_j}^n$ where I may appear inside the E_js only as a conclusion. This is called the *strict positivity* condition, and is mandatory for the system to be coherent [8];

$$\frac{}{\vdash \texttt{Prop} : \texttt{Type}_1}\ (\textsc{Ax}_1) \qquad \frac{}{\vdash \texttt{Set}_i : \texttt{Type}_{i+1}}\ (\textsc{Ax}_2) \qquad \frac{}{\vdash \texttt{Type}_i : \texttt{Type}_{i+1}}\ (\textsc{Ax}_3)$$

$$x \notin \Gamma, s \in \mathcal{S}\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ (\textsc{Var}) \qquad x \notin \Gamma, s \in \mathcal{S}\ \frac{\Gamma \vdash B : C \qquad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C}\ (\textsc{Weak})$$

$$B \equiv C, s \in \mathcal{S}\ \frac{\Gamma \vdash A : C \qquad \Gamma \vdash B : s}{\Gamma \vdash A : B}\ (\textsc{Conv}) \qquad B <: C\ \frac{\Gamma \vdash A : B}{\Gamma \vdash A : C}\ (\textsc{Cum})$$

$$r \in \{\texttt{Prop}, \texttt{Set}_i, \texttt{Type}_i\}\ \frac{\Gamma \vdash A : r \qquad \Gamma, x : A \vdash B : \texttt{Set}_i}{\Gamma \vdash \forall x : A.B : \texttt{Set}_i}\ (\forall_1)$$

$$r \in \{\texttt{Prop}, \texttt{Set}_i, \texttt{Type}_i\}\ \frac{\Gamma \vdash A : r \qquad \Gamma, x : A \vdash B : \texttt{Type}_i}{\Gamma \vdash \forall x : A.B : \texttt{Type}_i}\ (\forall_2)$$

$$s \in \mathcal{S}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : \texttt{Prop}}{\Gamma \vdash \forall x : A.B : \texttt{Prop}}\ (\forall_3)$$

$$\frac{\Gamma \vdash M : \forall x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B[N/x]}\ (\textsc{App}) \qquad \frac{\Gamma, x : A \vdash B : C}{\Gamma \vdash \lambda x : A.B : \forall x : A.C}\ (\textsc{Abs})$$

Figure 2 The refined calculus of construction with universes : $\mathsf{CIC_r}$

4. for any j, $\forall \overrightarrow{(z : E_j)}^{n_j}.I\,\overrightarrow{x}^p\,\overrightarrow{D_j}^n$ is a well-typed expression of sort r under the context $(\overrightarrow{(x : P)}^p, I : A)$.

Notice that we do not allow inductive definitions in a nonempty context, but this is only for a matter of clarity.

Declaring a new inductive definition adds new constants I and c_j to the system, together with the top left two typing rules presented in **Figure 3**.

The bottom rule of **Figure 3** is the typing rule for the `case` construction which is used to implement elimination schemes. Two sorts are involved in eliminations: s, the sort of the inductive type we eliminate, which may be `Prop` or `Set`; and r, the type of the type of the term we construct, which may be `Prop`, `Set` or `Type`. The four cases of elimination that do

$$\frac{}{\vdash I : A}\ (\textsc{Ind}) \qquad \frac{}{\vdash c_j : C_j}\ (\textsc{Constr}) \qquad f \text{ is guarded}\ \frac{\Gamma, f : A \vdash M : A}{\Gamma \vdash \texttt{fix}(f : A).M : A}\ (\textsc{Fix})$$

$$\text{(under restr.)}\ \frac{\begin{array}{c}\Gamma \vdash M : I\,\overrightarrow{Q}^p\,\overrightarrow{G}^n \qquad \Gamma \vdash T : \forall \overrightarrow{y : B[\overrightarrow{Q}^p/\overrightarrow{x}^p]}^n.I\,\overrightarrow{Q}^p\,\overrightarrow{y}^n \to r \\ \left(\Gamma \vdash F_j : \forall \overrightarrow{(z : E_j[\overrightarrow{Q}^p/\overrightarrow{x}^p])}^{n_j}.T\,\overrightarrow{D_j[\overrightarrow{Q}^p/\overrightarrow{x}^p]}^n\,(c_j\,\overrightarrow{Q}^p\,\overrightarrow{z}^{n_j})\right)_{j=1\ldots k}\end{array}}{\Gamma \vdash \texttt{case}_I(M, \overrightarrow{Q}^p, T, \overrightarrow{F}^k) : T\,\overrightarrow{G}^n\,M}\ (\textsc{Case})$$

Figure 3 The rules for an inductive type $\texttt{Ind}^p(I : A, c_1 : C_1, \ldots, c_k : C_k)$

not involve `Type` are called *small eliminations*. They are used to build:

- proofs and programs by inspecting programs;
- proofs by inspecting proofs;
- programs by inspecting proofs under restriction (1) (see below).

The other two cases are called *large eliminations*. Strong elimination is mainly used to build propositions by case analysis and to internally prove the minimality of informative inductive definitions. For instance, using large elimination over `nat`, one may build a predicate P of type $\mathtt{nat} \to \mathtt{Prop}$ such that $P\,0 \equiv \top$ and $P\,(\mathtt{S}\ 0) \equiv \bot$ and thus proves that $0 \neq (\mathtt{S}\ 0)$. Similarly, large elimination may be used to build informative types (for instance, to build a "type constructor" $T_{\alpha,\beta} : \mathtt{nat} \to \mathtt{Set}$ parametrized by a type α and an informative type β such that $T_{\alpha,\beta}\, n \equiv \alpha \to \cdots \to \alpha \to \beta$ with n occurrences of α) or to build arities (for instance, to build an "arity constructor" $A_\alpha : \mathtt{nat} \to \mathtt{Type}$ parametrized by a type α such that $A_\alpha\, n \equiv \alpha \to \cdots \to \alpha \to \mathtt{Prop}$ with n occurrences of α).

Eliminations from `Prop` to other sorts are restricted to inductive definitions that have at most one constructor, and such that all the arguments (which are not parameters) of this constructor are of sort `Prop`:

$$k = 0 \text{ or } \left(k = 1 \text{ and } \vdash E : \mathtt{Prop} \text{ for any } E \in \overrightarrow{E_1}^{n_1}\right) \tag{1}$$

This is essential for coherence [7] and has a computational interpretation: it is natural that computing an informative type should not rely on any proof structure, that would disappear during program extraction [15, 16].

▶ Example 1. Here are a few examples of inductive definitions:

- $\mathtt{Ind}^0(\mathtt{nat} : \mathtt{Set}_0,\quad 0 : \mathtt{nat},\quad \mathtt{S} : \mathtt{nat} \to \mathtt{nat})$
- $\mathtt{Ind}^1(\mathtt{list}_i : \mathtt{Set}_i \to \mathtt{Set}_i,\quad \mathtt{nil}_i : \forall A : \mathtt{Set}_i\,.\,\mathtt{list}_i\ A,\quad \mathtt{cons}_i : \forall A : \mathtt{Set}_i\,.A \to \mathtt{list}_i\ A \to \mathtt{list}_i\ A)$
- $\mathtt{Ind}^0(\mathtt{True} : \mathtt{Prop},\quad \mathtt{I} : \mathtt{True})$
- $\mathtt{Ind}^0(\mathtt{False} : \mathtt{Prop})$
- $\mathtt{Ind}^2(\mathtt{eq}_i : \forall (A : \mathtt{Set}_i).A \to A \to \mathtt{Prop},\quad \mathtt{refl}_i : \forall (A : \mathtt{Set}_i)(x : A).\,\mathtt{eq}_i\ x\,x)$
- $\mathtt{Ind}^2(\mathtt{eqP} : \forall (A : \mathtt{Prop}).A \to A \to \mathtt{Prop},\quad \mathtt{reflP} : \forall (A : \mathtt{Prop})(x : A).\,\mathtt{eqP}\ x\,x)$
- $\mathtt{Ind}^2(\mathtt{eqT}_i : \forall (A : \mathtt{Type}_i).A \to A \to \mathtt{Prop},\quad \mathtt{reflT}_i : \forall (A : \mathtt{Type}_i)(x : A).\,\mathtt{eqT}_i\ x\,x)$

Note that we have three levels of Leibniz equality: $\mathtt{eq}_i$ for comparing programs, `eqP` for comparing proofs and $\mathtt{eqT}_i$ for comparing everything else (we find the same kind of triplication for other standard encodings like cartesian product, disjoint sum and the existential quantifier).

The second operator to deal with inductive definitions is fixpoint definition. The typing rule for the fixpoint is defined on the top right of **Figure 3**. It is also restricted to avoid non-terminating terms, which would lead to absurdity. The restriction is called the *guard condition*: one argument should have an inductive type, and must structurally decrease on each recursive call. One may refer to [11] for further details.

We extend the reduction with the ι-reduction rules:

$$\begin{array}{rcl} \mathtt{case}_I(c_j\,\overrightarrow{Q}^p\,\overrightarrow{M}^{n_j}, \overrightarrow{Q}^p, T, \overrightarrow{F}^k) & \rhd & F_j\,\overrightarrow{M}^{n_j} \\ (\mathtt{fix}(f : A).M)\,(c_j\,\overrightarrow{Q}^p\,\overrightarrow{M}^{n_j}) & \rhd & M[\mathtt{fix}(f : A).M/f]\,(c_j\,\overrightarrow{Q}^p\,\overrightarrow{M}^{n_j}) \end{array}$$

and $\equiv$ denotes the $\beta\iota$-equivalence.

3.2 Embedding $\mathsf{CIC_r}$ into CIC and coherence

This calculus embeds easily into CIC by mapping $\mathtt{Set}_i$ and $\mathtt{Type}_i$ onto the sort $\mathtt{Type}_i$ of CIC:

▶ Lemma 1. Let $|\bullet|$ be the context-closed function from terms of $\mathsf{CIC_r}$ to terms of CIC such that $|\mathtt{Prop}| = \mathtt{Prop}$ and $|\mathtt{Type}_i| = |\mathtt{Set}_i| = \mathtt{Type}_i$, then we have :

$$\Gamma \vdash A : B \Rightarrow |\Gamma| \vdash_{\mathsf{CIC}} |A| : |B|$$

Since $|\forall X : \mathtt{Prop}.X| = \forall X : \mathtt{Prop}.X$, the logical coherence (the existence of an unprovable proposition) of CIC ensures the coherence of $\mathsf{CIC_r}$.

Conversely, some terms of CIC do not have a counterpart in the refinement: we cannot mix informative and uninformative types. An example is the following Coq definition:

```
fun (b:bool) ⇒ if b then nat else Set
```

4 Relational parametricity

In this setting, we have a natural notion of parametricity: we can define a translation that maps types to relations and other terms to proofs that they belong to those relations. What is new is that relations over objects of type Prop or Set have Prop as a codomain, which is more natural in a calculus with an impredicative sort for propositions.

We go step by step. First, we define parametricity for the calculus without inductive types, and show the abstraction theorem for this restriction. Subsequently, we add inductive types with large eliminations forbidden, and finally see how large eliminations behave with parametricity.

4.1 Parametricity for the calculus without inductive types

▶ Definition 1 (Parametricity relation). The parametricity translation $[\![\bullet]\!]$ is defined by induction on the structure of terms:

$$
\begin{aligned}
[\![\langle\rangle]\!] &= \langle\rangle && (1)\\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\, x\, x' && (2)\\
\\
[\![s]\!] &= \lambda(x : s)(x' : s).x \to x' \to \hat{s} && (3)\\
[\![x]\!] &= x_R && (4)\\
[\![\forall x{:}A.B]\!] &= \lambda(f : \forall x : A.B)(f' : \forall x' : A'.B').\quad \forall(x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\\
&\qquad\qquad [\![B]\!]\,(f\,x)\,(f'\,x') && (5)\\
[\![\lambda x : A.B]\!] &= \lambda(x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').[\![B]\!] && (6)\\
[\![(A\,B)]\!] &= ([\![A]\!]\,B\,B'\,[\![B]\!]) && (7)
\end{aligned}
$$

with $\hat{\mathtt{Prop}} = \hat{\mathtt{Set}}_i = \mathtt{Prop}$ and $\hat{\mathtt{Type}}_i = \mathtt{Type}_i$ and where A' denotes the term A in which we have replaced each variable x by a fresh variable x'.

It is easy to prove by induction that the previous definition is well-behaved with respect to substitution and conversion:

▶ Lemma 2 (Substitution lemmas). **1.** $(A[B/x])' = A'[B'/x']$
2. $[\![A[B/x]]\!] = [\![A]\!][B/x][B'/x'][[\![B]\!]/x_R]$
3. $A_1 \equiv_\beta A_2 \Rightarrow [\![A_1]\!] \equiv_\beta [\![A_2]\!]$

$$\Theta_I(\overrightarrow{Q}^p, T, \overrightarrow{F}^n) = \lambda\overrightarrow{(x:A)(x':A')(x_R:[\![A]\!]\,x\,x')}^n\;(a : I\,\overrightarrow{Q}^p\,\overrightarrow{x}^n)(a' : I\,\overrightarrow{Q'}^p\,\overrightarrow{x'}^n)$$
$$(a_R : [\![I]\!]\,\overrightarrow{Q\,Q'\,[\![Q]\!]}^p\,\overrightarrow{x\,x'\,x_R}^n\,a\,a').$$
$$[\![T]\!]\,\overrightarrow{x\,x'\,x_R}^n\,a\,a'\,a_R\;(\mathtt{case}_I\,(a, \overrightarrow{Q}^p, T, \overrightarrow{F}^n))\,(\mathtt{case}_I\,(a', \overrightarrow{Q'}^p, T', \overrightarrow{F'}^n))$$

Figure 4 The definition of Θ_I

The abstraction theorem states that the parametricity transformation preserves typing.

▶ Theorem 1 (Abstraction without inductive definitions). If $\Gamma \vdash A : B$, then $[\![\Gamma]\!] \vdash A : B$, $[\![\Gamma]\!] \vdash A' : B'$, and $[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,A\,A'$.

Proof. The proof is a straightforward induction on the derivation of $\Gamma \vdash A : B$. The first item is essentially proved by invoking structural rules and by propagating induction hypothesis. The key steps of the second items are the rule (Ax$_2$), which requires cumulativity, and the rules ($\forall_1$-$\forall_3$), which involve many abstraction and product rules. ◀

4.2 Why does not it work directly in CIC?

In the syntactic theory of parametricity for dependent types presented in [5], relations over a type of some universe are implemented as predicates ranging in the same universe. This can be read in the following piece of definition : $[\![\mathtt{Type}_i]\!] = \lambda(x\,x' : \mathtt{Type}_i).x \to x' \to \mathtt{Type}_i$. We cannot simply replace the conclusion with $\mathtt{Prop}$, because in CIC one has $\vdash \mathtt{Type}_i : \mathtt{Type}_{i+1}$, and the abstraction theorem would require that $\vdash [\![\mathtt{Type}_i]\!] : [\![\mathtt{Type}_{i+1}]\!]\;\mathtt{Type}_i\;\mathtt{Type}_i$ which is equivalent to $\vdash \lambda(x\,x' : \mathtt{Type}_i).x \to x' \to \mathtt{Prop} : \mathtt{Type}_i \to \mathtt{Type}_i \to \mathtt{Prop}$ but this last sequent is not derivable. In our refinement, $[\![\mathtt{Prop}]\!]$ and $[\![\mathtt{Set}]\!]$ have $\mathtt{Prop}$ as a conclusion, but this is not a problem since we do not have $\vdash \mathtt{Set}_i : \mathtt{Set}_{i+1}$.

This refined calculus is very convenient to set the basis for parametricity. As we argued, it has also nice properties regarding realizability and extraction: as an example, the correctness of extraction in this calculus would not rely on the termination of the β-reduction. Even if possible, obtaining the same result directly in CIC would have required a complete reworking of parametricity relations.

The calculus is very close to CIC, though. In Section 6, we discuss if it is possible to write a tactic in Coq that would exploit this work, without changing Coq's calculus.

4.3 Adding inductive types

As a first step, we restrict ourselves to small eliminations: we do not allow large eliminations. We will see in Subsection 4.4 that we are actually able to handle large eliminations over a big class of inductive definitions.

We write $\Gamma \vdash_{\mathrm{SE}} A : B$ to denote sequents typable in $\mathsf{CIC_r}$ where large eliminations are forbidden. Let us suppose that $\mathtt{Ind}^p(I : A, \overrightarrow{c : C}^k)$, we will define a fresh inductive symbol $[\![I]\!]$ and a family $([\![c_i]\!])_{i=1\ldots k}$ of fresh constructor names. Then we extend Definition 1 with

$$[\![\mathtt{fix}(x : A).B]\!] = (\mathtt{fix}(x_R : [\![A]\!]\,x\,x').[\![B]\!])\,[\mathtt{fix}(x : A).B/x][\mathtt{fix}(x' : A').B'/x']$$
$$[\![\mathtt{case}_I(M, \overrightarrow{Q}^p, T, \overrightarrow{F}^n)]\!] = \mathtt{case}_{[\![I]\!]}([\![M]\!], \overrightarrow{Q, Q', [\![Q]\!]}^p, \Theta_I(\overrightarrow{Q}^p, T, \overrightarrow{F}^n), \overrightarrow{[\![F]\!]}^n)$$

where Θ_I is defined in **Figure 4**.

We want to extend Theorem 1 with inductive definitions. We prove the following theorem:

▶ Theorem 2 (Abstraction with inductive definitions). **1.** If $\mathtt{Ind}^p(I : A, \overrightarrow{c : C}^k)$ is a valid inductive definition then so is $\mathtt{Ind}^{3p}(\llbracket I \rrbracket : \llbracket A \rrbracket\, I\, I, \overrightarrow{\llbracket c \rrbracket : \llbracket C \rrbracket\, c\, c'}^k)$.
2. If $\Gamma \vdash_{\mathrm{SE}} A : B$ then $\llbracket \Gamma \rrbracket \vdash_{\mathrm{SE}} A : B$, $\llbracket \Gamma \rrbracket \vdash_{\mathrm{SE}} A' : B'$, and $\llbracket \Gamma \rrbracket \vdash_{\mathrm{SE}} \llbracket A \rrbracket : \llbracket B \rrbracket\, A\, A'$.

Proof. The first item requires to check the constraints to build inductive types: the typing and the strict positivity. As for Theorem 1, the second item is proved by induction on the structure of the proof of $\Gamma \vdash A : B$. One needs to check that the guard condition is preserved in the (Fix) rule and that the (Case) rule is well-formed. The key idea here is that the translation of terms containing only small eliminations also contains only small eliminations. ◀

4.4 Overcoming the restriction over large elimination

Suppose we now authorize the whole large elimination (with restriction (1)). The definition generated by the following inductive definition $\mathtt{Ind}^0(\mathtt{box}_i : \mathtt{Set}_{i+1}, \mathtt{close}_i : \mathtt{Set}_i \to \mathtt{box}_i)$ is

$$\begin{aligned}\mathtt{Ind}^0 \big(&\llbracket \mathtt{box}_i \rrbracket : \mathtt{box}_i \to \mathtt{box}_i \to \mathtt{Prop},\\ &\llbracket \mathtt{close}_i \rrbracket : \forall (A\, A' : \mathtt{Set}_i).(A \to A' \to \mathtt{Prop}) \to \llbracket \mathtt{box}_i \rrbracket\, (\mathtt{close}_i\ A)\, (\mathtt{close}_i\ A')\big)\end{aligned}$$

If we want to prove parametricity for the (Case) rule when we build a $\mathtt{Type}$, one should provide an inhabitant of: $\forall (A\, A' : \mathtt{Set}_i).\llbracket \mathtt{box}_i \rrbracket\, (\mathtt{close}_i\ A)\, (\mathtt{close}_i\ A') \to (A \to A' \to \mathtt{Prop})$. But since $\llbracket \mathtt{box}_i \rrbracket\, (\mathtt{close}_i\ A)\, (\mathtt{close}_i\ A')$ has type $\mathtt{Prop}$ and $A \to A' \to \mathtt{Prop}$ has type $\mathtt{Type}$, we cannot build the expected relation by deconstructing a proof of $\llbracket \mathtt{box}_i \rrbracket\, (\mathtt{close}_i\ A)\, (\mathtt{close}_i\ A')$: this is forbidden by restriction (1).

However, let us consider the following example:

$$\mathtt{Ind}^0(I : \mathtt{Set}, \mathrm{N} : \mathtt{nat} \to I, \mathrm{B} : \mathtt{bool} \to I)$$

Let say we need to translate the following large elimination (for the sake of readability, we present it with the Coq syntax):

```
Definition f (x : I) := match x with
  | N n ⇒ vector n
  | B b ⇒ nat
end.
```

We can swap the destruction of x_R for two nested destructions of x and x' which produces k^2 branches (where k in the number of constructors). But only k of them are actually possible (we use here the Program keyword in order to let the system infer dependent type annotations for each match):

```
Program Definition f_R (x x' : I) (x_R : ⟦I⟧ x x') :=
  match x with
  | N n ⇒ match x' with
    | N n' ⇒ let n_R := inv n n' x_R in ⟦vector n⟧
    | B b' ⇒ absurd (vector n→nat→Prop) (abs₁₂ n b' x_R)
    end
  | B b ⇒ match x' with
    | N n' ⇒ absurd (nat→vector n'→Prop) (abs₂₁ b n' x_R)
    | B b ⇒ ⟦nat⟧
    end
  end.
```

where the following terms are all implemented with an authorized large elimination:

$$\begin{array}{rcl}
\texttt{inv} & : & \forall (n\,n' : \texttt{nat}).[\![I]\!]\,(\mathrm{N}\,n)\,(\mathrm{N}\,n') \to [\![\texttt{nat}]\!]\,n\,n' \\
\texttt{abs}_{12} & : & \forall (n : \texttt{nat})(b' : \texttt{bool}).[\![I]\!]\,(\mathrm{N}\,n)\,(\mathrm{B}\,b') \to \texttt{False} \\
\texttt{abs}_{21} & : & \forall (b : \texttt{bool})(n' : \texttt{nat}).[\![I]\!]\,(\mathrm{B}\,b)\,(\mathrm{N}\,n') \to \texttt{False} \\
\texttt{absurd} & : & \forall (\alpha : \texttt{Type}).\,\texttt{False} \to \alpha
\end{array}$$

We notice that this example runs smoothly because all the arguments of all the constructors have type Prop or Set, which avoids the pitfall of the box example.

That is why we propose to restrict large elimination from Set to Type to the class of small inductive definitions (this class was introduced by Paulin in [20] to restrict the large elimination in vanilla Coq where the sort Set of informative types is impredicative):

▶ Definition 2 (Small inductive definitions). We say that $\mathtt{Ind}^p(I : A, \overrightarrow{c : C}^k)$ is a *small inductive definition* if all the arguments of each constructor are of sort Prop or $\texttt{Set}_m$ for some m. More formally, if for all $1 \le i \le k$, $\vdash c_i : \forall \overrightarrow{(x : P)}^p \overrightarrow{(y : B)}^{n_i} .I\, \vec{x}^p\, \overrightarrow{D_j}^n$ then $\overrightarrow{x : P}^p, \overrightarrow{y : B}^{j-1} \vdash B_j : r$ with $r = \texttt{Prop}$ or $r = \texttt{Set}_m$ for some m.

With this restriction, the abstraction theorem holds in presence of large elimination:

▶ Theorem 3. Theorem 2 holds when $\vdash_{\mathrm{SE}}$ stands for derivability where large elimination is authorized over small inductive definitions and forbidden otherwise.

5 Examples of "free theorems"

In this section we give a few examples of consequences of the abstraction theorem. Most examples that can be found in the literature (see for instance [5, 26]) may be easily implemented in our framework. To improve readability, we use "$=_\alpha$" and "$\exists x : \alpha$" to denote respectively standard inductive encodings of the Leibniz equality and existential quantifier.

5.1 The type of Church numerals

Let $\texttt{church}_i$ be $\forall \alpha : \texttt{Set}_i, (\alpha \to \alpha) \to \alpha \to \alpha$, the type of Church numerals. Let $\texttt{iter}_i$ be the following expression

$$\begin{array}{l}
\texttt{fix iter}_i : \texttt{nat} \to \texttt{church}_i\, .\lambda (n : \texttt{nat})(\alpha : \texttt{Set}_i)(f : \alpha \to \alpha)(z : \alpha). \\
\qquad\qquad\qquad\qquad \texttt{case}(n, \lambda k : \texttt{nat}\, .\alpha, z, \lambda p : \texttt{nat}\, .f\, (\texttt{iter}_i\; p\, \alpha\, f\, z))
\end{array}$$

which is the primitive recursive operator which composes a function n times with itself.

The relation $[\![\texttt{church}_i]\!] : \texttt{church}_i \to \texttt{church}_i \to \texttt{Prop}$ is the relation unfolded in the introduction. One can prove easily the following property on any $f : \texttt{church}_i$:

$$[\![\texttt{church}_i]\!]\, f\, f \to\ \exists n : \texttt{nat}\, .\forall (\alpha : \texttt{Set}_i)(g : \alpha \to \alpha)(z : \alpha).\, \texttt{iter}_i\; n\, \alpha\, g\, z =_\alpha f\, \alpha\, g\, z$$

which states that, if f is in relation with itself by $[\![\texttt{church}_i]\!]$, then there exists an integer n such that f is extensionally equal to $\texttt{iter}_i\ n$. Now suppose we have a closed term F such that $\vdash F : \texttt{church}_i$. By the abstraction theorem we obtain a proof $[\![F]\!]$ that $[\![\texttt{church}_i]\!]\, F\, F$ and therefore that F is extensionally equal to $\texttt{iter}_i\ n$ for some n.

5.2 The tree monad

Binary trees carrying information of type α on their leaves may be implemented by the following inductive definition :

$$\mathtt{Ind}^1(\mathtt{tree}_i : \mathtt{Set}_i \to \mathtt{Set}_{i+1}, \mathtt{leaf}_i : \forall\alpha : \mathtt{Set}_i\,.\alpha \to T\,\alpha, \mathtt{node}_i : \forall\alpha : \mathtt{Set}_i\,.T\,\alpha \to T\,\alpha \to T\,\alpha)$$

and it is possible to represent in CIC the function $\mathtt{map}_i$ of type $\forall(\alpha\,\beta : \mathtt{Set}_i).(\alpha \to \beta) \to \mathtt{tree}_i\ \alpha \to \mathtt{tree}_i\ \beta$ which maps a function to all the leaves of a tree.

The generated relation $[\![\mathtt{tree}_i]\!]$ tells that two trees are related if they have the same shape and elements at the same position in each tree are related. It is then not difficult to prove for any function $f : \alpha \to \alpha'$ that $[\![\mathtt{tree}_i]\!]\,\alpha\,\alpha'\,R_f$ is a relation representing the graph of the $\mathtt{map}$ function where R_f is $\lambda(x : \alpha)(x' : \alpha').f x =_{\alpha'} x'$ and represents the graph of f.

We can also define in the system the multiplication of the monad by programming an expression μ_i of type $\forall\alpha.\,\mathtt{tree}_i\,(\mathtt{tree}_i\ \alpha) \to \mathtt{tree}_i\,\alpha$ with the following computational behavior:

$$\mu_i\,\alpha\,(\mathtt{leaf}_i\ \alpha\,x) \equiv x \qquad \text{and} \qquad \mu_i\,\alpha\,(\mathtt{node}_i\ \alpha\,x\,y) \equiv \mathtt{node}_i\ \alpha\,(\mu_i\,\alpha\,x)\,(\mu_i\,\alpha\,y)$$

As μ_i is closed, an application of the abstraction theorem which instantiates the relation to the graph of f proves the naturality of μ_i.

5.3 Parametricity and algebra

Obtaining "free theorems" by parametricity can be extended to data types with structure. In this section, we take the example of finite groups, which is directly related to the Ssreflect library [12] developed in Coq; but our reasoning applies to a large variety of algebraic structures.

In Chapter 3.4 of his PhD. thesis [9], François Garillot observed that algebraic developments require lots of proofs by isomorphism, which often look similar. Intuitively, a polymorphic function operating on groups can only compose elements using the laws given by the group's structure, and thus cannot create new elements.

More formally, we take an arbitrary group $\mathcal{H}$ defined by a carrier $\alpha : \mathtt{Set}_0$, a unit element $\mathtt{e} : \alpha$, a composition law $\cdot : \alpha \to \alpha \to \alpha$, an inverse function $\mathtt{inv} : \alpha \to \alpha$, and the standard axioms stating that $\cdot$ is associative, $\mathtt{e}$ is neutral on the left and composing with the inverse on the left produces the unit. On top of this, we define the type of all the finite subgroups of $\mathcal{H}$ with the following one-constructor inductive definition:

$$\begin{aligned}\mathtt{Ind}^0\Big(&\mathtt{fingrp} : \mathtt{Set}_0, \mathtt{Fingrp} : \forall\,\mathtt{elements} : \mathtt{list}\,\alpha.\\ &\quad \mathtt{e} \in \mathtt{elements} \to\\ &\quad (\forall x\,y.x \in \mathtt{elements} \to y \in \mathtt{elements} \to x \cdot y \in \mathtt{elements}) \to\\ &\quad (\forall x.x \in \mathtt{elements} \to \mathtt{inv}\,x \in \mathtt{elements}) \to \mathtt{fingrp}\Big)\end{aligned}$$

where $\in : \alpha \to \mathtt{list}\,\alpha \to \mathtt{Prop}$ is the standard inductive predicate stating if an element appears in a list.

Suppose we have a closed term $Z : \mathtt{fingrp} \to \mathtt{fingrp}$ (examples of such terms abound: eg. the center, the normalizer, the derived subgroup...). The abstraction theorem states that for any $R : \alpha \to \alpha \to \mathtt{Prop}$ compatible with the laws of $\mathcal{H}$ and for any $G\,G' : \mathtt{fingrp}$, $[\![\mathtt{fingrp}]\!]_R\,G\,G' \to [\![\mathtt{fingrp}]\!]_R\,(Z\,G)\,(Z\,G')$ where $[\![\mathtt{fingrp}]\!]_R$ is the relation on subgroups induced by R. Given this, we can prove the following properties:

- for any G, $Z\,G \subset G$ (if we take $R : x\,y \mapsto x \in G$);

- for any G, for any ϕ a morphism of $\mathcal{H}$, $\phi(Z\,G) = Z\,\phi(G)$ (if we take $R : x\,y \mapsto y = \phi(x)$). It entails that $Z\,G$ is a *characteristic subgroup* of $\mathcal{H}$.

To prove this, we use the axiom of proof irrelevance (that can be safely added to the system as we will show in the next subsection). The proof is straightforward by unfolding the definitions. A complete Coq script can be found online [1].

5.4 Classical axioms

One interesting feature of Coq is the ability to add axioms in the system. However when the parametricity transformation $[\![\cdot]\!]$ will encounter the axiom, it will ask for a proof that it is related to itself. Let consider an axiom P such that $\vdash P : s$ where s is Prop or Set. Here three situations are possible:

- Either P is what we call *provably parametric*: the user can provide a proof of $\forall h : P.[\![P]\!]\,h\,h$ and this proof may be used by the abstraction theorem to prove parametricity for terms involving the axiom.
- Or P is *provably not parametric*: there exists a proof that $\forall (h\,h' : P).\neg([\![P]\!]\,h\,h')$. It means that the axiom would break the parametricity of the system: there is no way to invoke the abstraction theorem on a term which uses that axiom.
- Or it is neither provably parametric nor provably not parametric or the user does not know. In this case, the parametricity of the axiom may be added as a new axiom at the user's risk.

Note that if $\neg P$ is provable then P is both provably parametric and provably not parametric and by the abstraction theorem, if P is provable then it is of course provably parametric. It is also easy to deduce from the abstraction theorem that if $P \to Q$ is provable then P provably parametric implies Q provably parametric, and Q provably not parametric implies P provably not parametric. Hence these notions do not depend on the formulation of your axioms.

5.4.1 Proof irrelevance

The axiom of *proof irrelevance* $\mathtt{PI} = \forall (X : \mathtt{Prop})(p\,q : X), p =_X q$ states that there is at most one proof of any proposition. It is provably parametric since

$$[\![\mathtt{PI}]\!]\,h\,h' = \forall (X\,X' : \mathtt{Prop})\,(X_R : X \to X' \to \mathtt{Prop})$$
$$(p : X)(p' : X')(p_R : X_R\,p\,p')(q : X)(q' : X')(q_R : X_R\,q\,q').[\![\mathtt{eqP}]\!]\,X\,X'\,p\,p'\,p_R\,q\,q'\,q_R$$

may be proved (with PI) equivalent to

$$\forall (X\,X' : \mathtt{Prop})\,(X_R : X \to X' \to \mathtt{Prop})(p : X)(p' : X')(p_R : X_R\,p\,p').$$
$$[\![\mathtt{eqP}]\!]\,X\,X'\,p\,p'\,p_R\,p\,p'\,p_R$$

which is directly provable by $[\![\mathtt{reflP}]\!]$. Therefore PI may be safely added to the system.

5.4.2 Independence of the law of excluded middle

From a user perspective provably not parametric axioms are bad news, but it provides meta-theoreticians a very simple way to prove independence results. Indeed, if a formula is provably not parametric then the abstraction theorem tells you this formula is not provable without large elimination over not small inductive definitions.

▶ Lemma 3. If P is provably not parametric, there is no closed term A of type P (in the restriction of large elimination to small inductive definitions).

For instance, Peirce's law $\texttt{Peirce} = \forall(X\,Y : \texttt{Prop}).((X \to Y) \to X) \to X$ (which is known to be equivalent to the excluded middle) is provably not parametric.

6 Towards a Coq implementation

This paper sets the theoretical foundation for an implementation of a reflexive Coq tactic generating the consequences of parametricity for definitions in the Calculus of Constructions. Two approaches are possible:

- modify Coq's calculus to implement $\mathsf{CIC_r}$. The implementation of the translation becomes straightforward;
- do not modify Coq's calculus, but let the translation distinguish informative terms.

The first approach would require to transform Coq radically. We followed the second approach, and started the implementation of a prototype for Coq commands and tactics for parametricity, called CoqParam [1].

In a system like Coq, *reflection* establishes a correspondence between:

- a subset of the Coq terms: this is called the *shallow embedding*;
- a Coq inductive data type representing these terms: this is called the *deep embedding*;
- the OCaml internal representation of those terms.

The deep embedding and the OCaml representation give access to the structure of the terms (whereas the shallow embedding does not), which is very useful to build properties and proofs by computing over this structure. This process, called *computational reflection*, is a well-known way to design powerful automatic tactics in Coq [2,13,14].

Parametricity comes well within the spirit of computational reflection: the abstraction theorem is a way to build proofs of terms by inspecting their structures. Our tactic is based on this remark: given a well-typed closed term $\vdash A : B$, it builds the well-typed proof $\vdash [\![A]\!] : [\![B]\!]\,A\,A$, going from the shallow embedding to the OCaml internal representation (this step is called *reification*), and the other way round. The difficulty is to decide, during reification, whether objects of type $\texttt{Type}$ in Coq should have type $\texttt{Set}$ or $\texttt{Type}$ in $\mathsf{CIC_r}$. The tactic does not handle this yet (as well as full inductive types).

Notice that, with this method, we do not have to generally prove the abstraction theorem in Coq: Coq's type checker will prove it on each instance. One may also be interested in a formal proof of the abstraction theorem. It means that the deep embedding should be defined. As the refinement is very close to Coq, this would thus require a large effort.

7 Related works and discussion

Since the introduction of parametricity for system F [22,26], it has been extended to many logical systems based on Type Theory. Among others, we can cite system $\mathcal{F}_\omega$ by Vytiniotis and Weirich [25] and a large subset of PTSs by Bernardy *et al.* [5,6]. In all these presentations, no sort is impredicative, and parametricity relations live either in a meta-logic or in a different sort than propositions. To our knowledge, this is the first time parametricity relations live in an impredicative sort representing propositions, making them more usable in a system like Coq.

Bernardy *et al.* [5] also explain two possible ways to handle inductive definitions: one by translating induction principles, and one by defining a new inductive data-type as the translation of the initial data-type. Our approach is close to the second method proposed by [5]. We also show how to translate fixpoint definitions, which are more common than inductive principles.

Parametricity and parts of the abstraction theorem have been formalized for deep embeddings of logical systems in Agda [5] and in Coq [3, 4]. Our approach is different: we do not want to have a formal proof of the abstraction theorem (in a first step), but we want to have a practical tool that actually computes results produced by the abstraction theorem. This does not compromise soundness anyway, since the terms produced by this tool are type-checked by Coq's kernel.

8 Conclusion

As we argue throughout the article, the system presented here distinguishes clearly via typing which expressions will be computationally meaningful after extraction. It allows us to define a notion of parametricity for which relations lie in the sort of propositions. This opens up a new way to define automatic tactics in interactive theorem provers based on Type Theory.

Moreover it is known that parametricity and realizability seen as syntactic constructions are closely related [6]. That is why it seems possible to build an internal realizability theory inside our framework. It would permit to develop a similar tactic to prove automatically that program extracted from any closed term will realize its own type. The user would then be able to use this proof to show the correctness of his programs without relying on the implementation of the extraction function.

Finally, it remains to understand why parametric relations do not fit in the sort of proposition in presence of large elimination on non-small data types. We conjecture that parametric relations for large inductive definitions are not proof-irrelevant (in particular, they cannot be interpreted as set-theoretical relations).

Acknowledgments The authors are particularly grateful to François Garillot and Georges Gonthier who suggested the use of parametricity to obtain theorems from free in the setting of algebra, and provided the stimulus for this work. We also thank Assia Mahboubi for providing useful help about the spirit of the Ssreflect library. We finally thank the anonymous reviewers for their encouragements and constructive remarks.

References

1 Preliminary implementation of a Coq tactic. `http://www.lix.polytechnique.fr/~keller/Recherche/coqpara`

2 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

3 Robert Atkey. A deep embedding of parametric polymorphism in Coq. In *Workshop on Mechanizing Metatheory*, 2009.

4 Robert Atkey. Syntax for Free: Representing Syntax with Binding Using Parametricity. In Pierre-Louis Curien, editor, *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.

5 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 345–356. ACM, 2010.

6 Jean-Philippe Bernardy and Marc Lasson. Realizability and Parametricity in Pure Type Systems. In Martin Hofmann, editor, *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2011.

7 Thierry Coquand. An Analysis of Girard's Paradox. In *LICS*, pages 227–236. IEEE Computer Society, 1986.

8 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

9 François Garillot. *Generic Proof Tools and Finite Group Theory*. PhD thesis, École Polytechnique, 2011.

10 Herman Geuvers. Inconsistency of classical logic in type theory. Unpublished notes, 2001.

11 Eduardo Giménez. Codifying Guarded Definitions with Recursive Schemes. *Types for proofs and Programs*, pages 39–59, 1995.

12 Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A Modular Formalisation of Finite Group Theory. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.

13 Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005.

14 Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A Computational Approach to Pocklington Certificates in Type Theory. *Functional and Logic Programming*, 2006.

15 Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.

16 Pierre Letouzey. *Programmation fonctionnelle certifiée: L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

17 Pierre Letouzey. Extraction in Coq: An Overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

18 Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers Univ. of Tech, 2007.

19 Christine Paulin-Mohring. Extracting F(omega)'s Programs from Proofs in the Calculus of Constructions. In *POPL*, pages 89–104, 1989.

20 Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In Marc Bezem and Jan Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer Berlin / Heidelberg, 1993. 10.1007/BFb0037116.

21 Gordon D. Plotkin and Martín Abadi. A Logic for Parametric Polymorphism. In Marc Bezem and Jan Friso Groote, editors, *TLCA*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1993.

22 John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.

23 Izumi Takeuti. An Axiomatic System of Parametricity. *Fundam. Inform.*, 33(4), 1998.

24 The Coq Development Team. *The Coq Proof Assistant: Reference Manual*. INRIA, 2012.

25 Dimitrios Vytiniotis and Stephanie Weirich. Parametricity, Type Equality, and Higher-Order Polymorphism. *Journal of Functional Programming*, 20(02):175–210, 2010.

26 Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

27 Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theor. Comput. Sci.*, 375(1-3):201–226, 2007.