Information Retrieval in a Coq Proof Library using Type Isomorphisms

David Delahaye*

Project Coq
INRIA-Rocquencourt**

Abstract. We propose a method to search for a lemma in a Coq proof library by using the lemma type as a key. The method is based on the concept of type isomorphism developed within the functional programming framework. We introduce a theory which is a generalization of the axiomatization for the simply typed λ-calculus (associated with Closed Cartesian Categories) to an Extended Calculus of Constructions with a more Extensional conversion rule. We show a soundness theorem for this theory but we notice that it is not contextual and requires "ad hoc" contextual rules. Thus, we see how we must adapt this theory for Coq and we define an approximation of the contextual part of this theory, which is implemented in a decision procedure.

1 Introduction

The problem of easily finding software components in a library is fundamental. It is connected to code reusability. Indeed, a reusable code is one which is not only sufficiently generic but one which can also be found quickly when needed[1]. This second component is often neglected because it is considered, wrongly, not to be very theoretical. Consequently, most current search tools are nothing more than identifiers indexes in which we hope systematically that the name given to the required function is sufficiently explicit for it to be found quickly. If you are the single author of the library you scan, the speed of your search depends only on your own memory, but if you are a co-author or not an author at all then the task may be very tedious. Thus and in a general way, we waste time in this approximate search which, if it fails, obliges the user to write code which may already exist. A typical example is that of the Caml function `list_it`[2] which, as shown in table 1, has four different names in other ML versions.

* David.Delahaye@inria.fr, http://coq.inria.fr/~delahaye/.

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

[1] In this regard, A. Mili, R. Mili and R. T. Mittermeir give a broad survey of software storage and retrieval methods in [15], where "software" is not necessarily only executable code.

[2] This is an abbreviated version of an example originally due to Mikael Rittri in [16].

Language	Name	Type
LCF ML ([8])	itlist	$\forall\alpha\beta.(\alpha \to \beta \to \beta) \to List(\alpha) \to \beta \to \beta$
Caml ([12])	list_it	$\forall\alpha\beta.(\alpha \to \beta \to \beta) \to List(\alpha) \to \beta \to \beta$
Haskell ([9])	foldr	$\forall\alpha\beta.(\alpha \to \beta \to \beta) \to \beta \to List(\alpha) \to \beta$
SML of New Jersey ([3])	fold	$\forall\alpha\beta.(\alpha \times \beta \to \beta) \to List(\alpha) \to \beta \to \beta$
Edinburgh SML ([5])	fold_right	$\forall\alpha\beta.(\alpha \times \beta \to \beta) \to \beta \to List(\alpha) \to \beta$

Table 1. The `list_it` in Caml.

As can be seen, an identifier is totally insufficient to allow a powerful search. The idea is thus to take the type as a search pattern and to carry out comparisons modulo a certain equivalence. Then, the following question arises: when are two types equivalent? There is no standard answer to this question. It depends on what we want to identify. A first naive choice could be to take syntactic equality. But this option is too restrictive as shown again by the example of `list_it` in Caml (see table 1) which has four distinct types in the various ML. So, the equivalence must be broader. Some work by Mikael Rittri ([16]) highlighted that the most favourable concept for search in libraries is that of isomorphic types. This concept, formalized and studied for many years by Roberto Di Cosmo, Giuseppe Longo, Kim Bruce and Sergei Soloviev (see, for example, [17], [6] and [7]), was implemented in a tool called CamlSearch, developed by Jerôme Vouillon and Julien Jalon ([18]) at the LIENS in 1994. CamlSearch extends the theory used by Mikael Rittri (typically the seven axioms for Closed Cartesian Categories) to polymorphism and deals with unification. Another study by Maria-Virginia Aponte, Roberto Di Cosmo and Catherine Dubois ([1], [2]) tries to include the modules of Objective Caml ([13]). The objective of this work is to make such a tool for Coq[3] ([4]).

First of all, we present the problem framework. Next, we see the basic concepts relating to type isomorphisms in order to build, thereafter, a theory in a type theory with extensional rules. From there, we adapt this theory to Coq and we define a decision procedure. Finally, we discuss our implementation and we provide some examples which give an idea about its use and its performances.

2 Framework

2.1 Extensions

The idea is to extend the theories built for programming languages to type theory. We do not aim to capture all type categories but only those which may be useful. So, it is not only a practical approach in the field of logic but also a theoretical study.

[3] From another perspective, Thomas Kolbe and Christoph Walther propose, in [10] and [11], to generalize computed proofs to produce schemes (for proofs and lemmas) which can match new lemmas and so can be reused.

An initial and easy extension is to have free polymorphism. Indeed, CamlSearch can only deal with an ML-polymorphism (quantifiers on type variables appear only in the head to make type inference decidable). This restriction has to be lifted and, for example, if we look for a Gödel recursor for type T, we expect that the two following types can be identified:

$$T \rightarrow (nat \rightarrow T \rightarrow T) \rightarrow nat \rightarrow T$$
$$nat \rightarrow T \rightarrow (nat \rightarrow T \rightarrow T) \rightarrow T$$

Next, it is quite natural to deal with dependent types although it makes the problem be much more difficult for reasons which will become clear. In this task, we must be careful with variable renamings and especially with variable bindings. A typical example with dependent types could be the following lemma on integers:

$$\forall n, m, p, q : Z.n \leq m \rightarrow p \leq q \rightarrow (n + p) \leq (m + q)$$

The user may want to move the variables p and q to the right of the subterm $n \leq m$ (because they do not occur) and so, he/she can give the following type to hit the previous lemma:

$$\forall n, m : Z.n \leq m \rightarrow \forall p, q : Z.p \leq q \rightarrow (n + p) \leq (m + q)$$

In addition to dependent products, we may want to capture dependent tuples. Usual existential quantifiers are concerned but, in a theory with primitive inductive types, we can define other existential quantifiers, which are inductive types with one constructor and without a recursive position. For example, we can specify the Euclidean division using the following inductive type:

$$\begin{array}{l} type\ diveucl\ (a, b : nat) = \\ \qquad divex : \forall q, r : nat.b > r \rightarrow a = (q * b) + r \end{array}$$

Where *divex* is the single constructor of *diveucl*. Then, the theorem is expressed as follows:

$$\forall b : nat.b > 0 \rightarrow \forall a : nat.(diveucl\ a\ b)$$

Here, *diveucl* plays the role of a customized existential quantifier, and the user, who does not know the existence of *diveucl*, certainly expects to find the theorem by using the usual existential quantifiers to express his/her type, which may be:

$$\forall a, b : nat.b > 0 \rightarrow \exists q, r : nat.(a = (q * b) + r) \wedge (b > r)$$

2.2 Limitations

In the present work, there are some features we do not want to deal with and this leads to limitations which we must identify.

First of all, the user has to know the vocabulary or a part of the vocabulary used for the semantical notions he/she wants to look for. For example, if he/she wants to search for theorems on natural numbers, he/she has to know that the keyword for natural numbers is `nat` and not `N` or `natural`; likewise, he/she has to be careful with the operators on natural numbers like the addition which is `plus` and not `plus_nat` or `nat_plus`. So, some queries may be difficult without an oracle which gives signs to the user to express correctly his/her formulae. This task could be made easier with a vocabulary system (like in Mizar) extracted from each module (typically a file in Coq) and combined with a command which gives the list of the modules.

We want to avoid constant expansion. The reason is that we want to keep an acceptable level of complexity. We must not forget that this procedure may be applied to large developments (industrial or mathematical) where complexity must be contained.

For the same reason, we dismiss the possibility of congruences. However, we agree that unpleasant surprises may occur without this option. There are many examples which show that congruences are important but we can choose a case where equality occurs e.g. the associativity of real numbers:

$$\forall r1, r2, r3 : R.(r1 + r2) + r3 = r1 + (r2 + r3)$$

Without symmetry on equality, if the user inverts the two members of the equality in his/her search then he/she will not find anything. Equality is not the only operator we may want to deal with, structure operators like addition or propositional operators like conjunction are included within this context.

We do not want to deal with pattern-matching here. That must be the result of another study. From this possibility, we may expect that, in the context where A, B, C and D are propositional variables, the type $A \rightarrow \Omega \rightarrow D$ where Ω is a metavariable can capture the following types:

$$\begin{array}{l} A \rightarrow B \wedge C \rightarrow D \\ A \rightarrow B \rightarrow C \rightarrow D \\ A \rightarrow C \rightarrow B \rightarrow D \end{array}$$

In a general way, parts of lemmas could be forgotten or hidden and this facility would certainly be used very often.

Finally, we do not want to capture general inductive types. Just as for pattern-matching, we consider that it must be the subject of another study, which can only be an abstraction with respect to the constructor names or something more semantic.

3 Isomorphisms of types

This section gives the basic definitions and concepts, mostly formalized in [7]. From these notions, it is possible to build a theory (axiomatization) which characterizes a certain class of isomorphisms for a given language.

In a natural way, we can say that two types are isomorphic if there exist two functions of conversion, which are definable in the reference language and which allow us to pass from one type to the other one and vice versa. More precisely, we have the following definition:

Definition 1 (Definable isomorphisms, invertible terms). *Two types A and B are definably isomorphic ($A \cong_d B$) if and only if there exist the functions (λ-terms) $M : A \to B$ and $N : B \to A$ such that $M \circ N =_{\mathcal{L}} \lambda x : B.x$ and $N \circ M =_{\mathcal{L}} \lambda x : A.x$ where $=_{\mathcal{L}}$ is an equality over the terms. The terms M and N are said to be invertible.*

This definition is parametrized by the choice of $=_{\mathcal{L}}$ which depends on the isomorphisms we want to deal with. In general, $=_{\mathcal{L}}$ contains $\beta\eta$-convertibility, projections, surjective-pairing and *void* substitution for terms in *unit* (terminal object).

Beyond this syntactic concept of type isomorphisms, we can obtain a more semantic view by considering the models of the language. Thus, two types are isomorphic in a specific model $\mathcal{M}$ if their interpretations are isomorphic in $\mathcal{M}$, in the traditional sense (i.e., there exist, in the model, two invertible functions f and g between them). Two types are semantically isomorphic if they are isomorphic for every model of the calculus.

There are many languages where the two notions of isomorphisms correspond. [7] gives some examples like the simply typed λ-calculus, the system F or the simply typed λ-calculus with Cartesian product and/or *unit*.

4 Formalism and theory

In the above, the definitional equality $=_{\mathcal{L}}$ is essentially β-conversion extended by more extentional simplifications like η-conversion or surjective pairing. Up to now, the main reason for considering these additional reductions was to make the theory complete, that is to have the syntactical and semantical notions coincide. In the case of calculi with dependent types, the situation changes. Namely, these generalized η-reductions become necessary not only for a matter of completeness, but also, more drastically, to be able to build up a theory compatible with the typing.

In this section, we will point out two difficulties due to the presence of dependent types:

1. In order to define the syntactic notion of isomorphism, we will have to explicitly keep track of the conversion function. For instance, given two types A and B and the corresponding functions $\sigma : A \to B$ and $\tau : B \to A$, it makes no

sense to consider an isomorphism between $\Pi x : A.C$ and $\Pi x : B.C$; there is no reason for having both these two types well formed at the same time. However, we can exhibit functions between $\Pi x : A.C$ and $\Pi x : B.C[x \leftarrow (\tau\ x)]$.
2. Furthermore, in the general case, the condition that σ and τ commute appears also to be necessary in order to build well typed equations.

Thus, this section is devoted to the presentation of a generalization of the definition of type isomorphisms for an Extended Calculus of Constructions with a more Extensional conversion rule (ECCE for short).

4.1 Definition of ECCE

ECCE is an extension of the Calculus of Constructions with predicative universes, Σ-types, *unit* and extensionality rules. The terms of ECCE can be inductively defined as the smallest set verifying the following clauses:

- $Prop$ and $Type_i$ with $i \in \mathbb{N}$ are terms;
- Variables are terms;
- *unit* and () are terms;
- If A, B, M and N are terms then $\Pi x : A.B$, $\Sigma x : A.B$, $\lambda x : A.M$, $(M, N)_{\Sigma x:A.B}$, $(\pi_1\ M)$, $(\pi_2\ M)$ and $M\ N$ are terms.

Terms are identified modulo α-conversion. We also denote $\Pi x : A.B$ and $\Sigma x : A.B$, respectively $A \to B$ and $A \times B$ when $x \notin B$. Reduction ($\to$) and conversion ($\simeq$) are defined as usual from the following one-step rules:

$$(\lambda x : A.M)\ N \to_1 M[x \leftarrow N] \qquad (\beta)$$
$$\lambda x : A.M\ x \to_1 M \text{ if } x \notin M \qquad (\eta)$$
$$(\pi_i\ (M_1, M_2)_{\Sigma x:A.B}) \to_1 M_i\ (i = 1, 2) \qquad (P)$$
$$((\pi_1\ M), (\pi_2\ M))_{\Sigma x:A.B} \to_1 M \qquad (SP)$$
$$M \to_1 () \text{ if } \Gamma \vdash M : unit \qquad (U)$$

$Prop$ and $Type_i$ are called universes and there exists an inclusion between them. This type inclusion induces a type cumulativity characterized by the partial order $\preceq$ over the terms, which is the smallest relation such that:

- $Prop \preceq Type_0 \preceq Type_1 \preceq ...$;
- if $A \preceq A'$ and $B \preceq B'$ then
  $\Pi x : A.B \preceq \Pi x : A'.B'$ and $\Sigma x : A.B \preceq \Sigma x : A'.B'$.

Typing contexts are lists of expressions of the form $x : A$ where x is a variable and A is a term. The empty context is the empty list noted []. A judgement is either Γ is well formed or $\Gamma \vdash t : T$ where Γ is a context and, t and T are terms. $FV(\Gamma)$, where Γ is a context of the form $[x_0 : A_0; ...; x_i; A_i; ...; x_n : A_n]$, denotes the union of x_i and $FV(A_i)$.

The inference rules of ECCE are given in appendix A. The term M is well typed under Γ if and only if $\Gamma \vdash M : A$ is derivable for an A.

ECCE can be seen as an extension of the Extended Calculus of Constructions (ECC) [14] with *unit* and extensionality rules.

4.2 Equations

Now, we can give equations over the terms of ECCE which are considered to be valid isomorphisms of ECCE. These equations deal with properties about function types, Σ-types and *unit*, which we used to handle, in a non-dependent way, in functional programming. If A, B and C are terms then the equations are the following:

1. $A = B$ if $A \simeq B$
2. $\Sigma x : A.B = \Sigma x : B.A$ if $x \notin FV(A, B)$
3. $\Sigma x : (\Sigma y : A.B).C = \Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)]$
4. $\Pi x : (\Sigma y : A.B).C = \Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]$
5. $\Pi x : A.\Sigma y : B.C = \Sigma y : (\Pi x : A.B).\Pi x : A.C[y \leftarrow (y\ x)]$
6. $\Sigma x : A.unit = A$
7. $\Sigma x : unit.A = A[x \leftarrow ()]$
8. $\Pi x : A.unit = unit$
9. $\Pi x : unit.A = A[x \leftarrow ()]$

This system is called $\mathsf{Ax}^{\mathsf{ECCE}}$. We consider only well typed types, that is to say, for Γ, if $A = B$ is an instance of an equation of $\mathsf{Ax}^{\mathsf{ECCE}}$ then $\Gamma \vdash A : s$ and $\Gamma \vdash B : s$, where $s \in \mathcal{S}$ with $\mathcal{S}$, the set of the universes.

If we exclude the axioms 1 (conversion) and 7 (() substitution when the witness of a Σ-type is of type *unit*), and if we ignore dependencies, we can recognize the seven axioms that Sergei Soloviev proved complete for Closed Cartesian Categories [17].

4.3 Theory

With dependencies, we guess that the theory built on $\mathsf{Ax}^{\mathsf{ECCE}}$ cannot be contextual, i.e. if we consider the terms T, A and B where A is a subterm of T and where $A = B$, then we do not always have $T = T'$ where T' is the term T for which B is substituted for some occurrences of A. Indeed, in $\mathsf{Ax}^{\mathsf{ECCE}}$, for Γ, if the left member is well typed (under Γ) then the right member will be too. This property is not valid for the contextual closure, i.e. the relation including the axioms of $\mathsf{Ax}^{\mathsf{ECCE}}$ and which, given the terms T, T', A, B, contains couples (T, T') such that $A = B$, A is a subterm of T and T' is the term T where B is substituted for some occurrences of A. To show why, let us define the notion of term context:

- terms are contexts;
- [] is a context;
- If A, B, M and N are contexts then $\Pi x : A.B$, $\Sigma x : A.B$, $\lambda x : A.M$, $(M, N)_{\Sigma x:A.B}$, $(\pi_1\ M)$, $(\pi_2\ M)$ and $M\ N$ are contexts.

If $\mathcal{C}$ is a context and A a term, $\mathcal{C}[A]$ denotes the term $\mathcal{C}$ where A is subsituted for []. A is called the argument of $\mathcal{C}$.

Now, let us consider the following context:

$$\mathcal{C} = \Pi f : [].\Pi c : (\Sigma y : A.B).(f \;\; c)$$

With $\mathcal{C}$ and the terms A, B and C, and if we suppose that, for Γ, $\Gamma \vdash A : Type_i$, $\Gamma, y : A \vdash B : Type_i$ and $\Gamma, x : (\Sigma y : A.B) \vdash C : Type_i$ are derivable then we can build the term $\mathcal{C}[\Pi x : (\Sigma y : A.B).C]$ which is well typed (under Γ). Using axiom 4 (curryfication) of $\mathsf{Ax}^{\mathsf{ECCE}}$ on the argument of $\mathcal{C}$, we notice that the resulting term is not well typed (under Γ).

To preserve typing, we have to modify $\mathcal{C}$ as follows:

$$\mathcal{C}' = \Pi f : [].\Pi c : (\Sigma y : A.B).(f \;\; (\pi_1 \;\; c) \;\; (\pi_2 \;\; c))$$

With $\mathcal{C}'$, the following equation is valid (deduced by $\mathsf{Ax}^{\mathsf{ECCE}}$) and well typed:

$$\mathcal{C}[\Pi x : (\Sigma y : A.B).C] = \mathcal{C}'[\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]]$$

$\mathcal{C}$ has been modified in the following way:

$$\begin{aligned} \mathcal{C}' &= \Pi f : [].\Pi c : (\Sigma y : A.B).(f \; (\pi_1 \; c) \; (\pi_2 \; c)) \\ &= \Pi f : [].\Pi c : (\Sigma y : A.B). \\ &\quad (((\lambda f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]).\lambda x : (\Sigma y : A.B). \\ &\quad f \; (\pi_1 \; x) \; (\pi_2 \; x)) \; f) \; c) \\ &= \Pi f : [].\Pi c : (\Sigma y : A.B).((\tau \;\; f) \;\; c) \end{aligned}$$

Where τ is the invertible term from the right-hand term to the left-hand one of axiom 4 of $\mathsf{Ax}^{\mathsf{ECCE}}$.

This example gives us several indications. Some contexts cannot be crossed without modifications and these modifications involve invertible terms. Thus, to build the theory over $\mathsf{Ax}^{\mathsf{ECCE}}$, which will be called $\mathsf{Th}^{\mathsf{ECCE}}$, we have to justify syntactically the axioms of $\mathsf{Ax}^{\mathsf{ECCE}}$, that is to say, to give the associated invertible terms, and we have to define dedicated contextual inference rules to make the equations applicable to the contexts.

We use the following notation:

$$\underset{\binom{\sigma}{\tau}}{A = B}$$

Where, for Γ, $\Gamma \vdash \sigma : A \rightarrow B$ and $\Gamma \vdash \tau : B \rightarrow A$. We also note this equation as follows: $\sigma : A = B : \tau$.

For instance, axiom 4 of $\mathsf{Ax}^{\mathsf{ECCE}}$ will be completed as follows:

$$\underset{\binom{\lambda f : (\Pi x : (\Sigma y : A.B).C).\lambda x : A.\lambda y : B[y \leftarrow x].f \; (x, y)}{\lambda f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]).\Pi x : (\Sigma y : A.B).f \; (\pi_1 \; x) \; (\pi_2 \; x)}}{\Pi x : (\Sigma y : A.B).C = \Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]} \quad (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{Cur}})$$

And, to cross Π-expressions to the left, we have:

$$\frac{\underset{\binom{\sigma}{\tau}}{A = A'}}{\underset{\binom{\lambda f : (\Pi x : A.B).\lambda x : A'.f \; (\tau \; x)}{\lambda f : (\Pi x : A'.B[x \leftarrow (\tau \; x)]).\lambda x : A.f \; (\sigma \; x)}}{\Pi x : A.B = \Pi x : A'.B[x \leftarrow (\tau \; x)]}} \quad (\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{L}})$$

The other axioms and inference rules of $\mathsf{Th}^{\mathsf{ECCE}}$ are given in appendix B. For A, B, σ and τ, four terms, the equation $\sigma : A = B : \tau$ is valid, which is noted $\Gamma \vdash \sigma : A = B : \tau$ if and only if $\Gamma \vdash A : s$, $\Gamma \vdash B : s$, with $s \in \mathcal{S}$, and $\sigma : A = B : \tau$ is derivable. For two terms A and B, an equation $A = B$ is valid, which is noted $\Gamma \vdash A = B$, if there exist σ and τ such that $\Gamma \vdash \sigma : A = B : \tau$.

Regarding the deduction rules, we have to notice that there is no rule for contexts which are λ-expressions, pairs or applications because these contexts cannot be crossed.

4.4 Soundness

The soundness theorem of $\mathsf{Th}^{\mathsf{ECCE}}$ is expressed as follows:

Theorem 1 (Soundness). *If $\Gamma \vdash \sigma : A = B : \tau$ then $\Gamma \vdash \sigma : A \to B$, $\Gamma \vdash \tau : B \to A$, $\sigma \circ \tau \simeq \lambda x : B.x$ and $\tau \circ \sigma \simeq \lambda x : A.x$.*

Proof. By induction on the derivation of $\Gamma \vdash \sigma : A = B : \tau$.

5 Adaptation to Coq

5.1 Restriction

To use $\mathsf{Th}^{\mathsf{ECCE}}$ in Coq, a natural way consists of getting rid of the extensional rules in reduction and conversion. Indeed, if we leave to one side δ-reduction (expansion of constants in an environment) and ι-reduction (reduction for primitive inductive types), Coq is only concerned by β-reduction. Also, we have to adapt the soundness theorem by substituting conversion with extensional rules for conversion (in this case, we use isomorphisms proved outside the formalism).

This restriction is not strong enough to ensure the soundness theorem. For example, let us consider the rule $\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{L}}$. We suppose we have Γ such that $\Gamma \vdash \sigma : A = A' : \tau$ and such that $\Pi x : A.B$ and $\Pi x : A'.B[x \leftarrow (\tau\ x)]$ are of type s with $s \in \mathcal{S}$. The second invertible term of the conclusion is of type $\Pi x : A'.B[x \leftarrow (\tau\ x)] \to \Pi x : A.B[x \leftarrow (\tau\ (\sigma\ x))]$ and the term $(\tau\ (\sigma\ x))$ cannot always be reduced to x due to the absence of extensional rules. So, some invertible terms are not of the expected type and this invalidates the soundness theorem.

The problem is that invertible terms can appear in types. One solution is to prevent such occurences, that is to say, to take out the rules $\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{L}}$ and $\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{L}}$ except for $\mathsf{Th}^{\mathsf{ECCE}}_{\beta}$ which creates reductible redexes.

So, this means that, to implement $\mathsf{Th}^{\mathsf{ECCE}}$ in Coq, we must consider a subset of this theory with a conversion rule based only on β-reduction and with a restriction on the use of rules which introduce invertible terms in types.

5.2 Σ-types and *unit*

In Coq, we consider Σ-types as inductive types with one constructor without a recursive position and without any constraint on their parameters. *unit* is any inductive type with one empty constructor.

5.3 A decision procedure

To implement $\mathsf{Th}^{\mathsf{ECCE}}$ for Coq, we extract a rewriting system, called $\mathcal{R}^{Coq}$, from the theory[4]. The rules of $\mathcal{R}^{Coq}$ are given in appendix C.

An immediate consequence of the previous restriction is that $\mathcal{R}^{Coq}$ is not confluent. For example, if A and B are terms, the following critical pair cannot be reduced:

$$\begin{array}{ccc}
 & \Pi x : unit.\Sigma y : A.B & \\
(\mathcal{R}^{Coq}_{\Pi Unl}) \swarrow & & \searrow (\mathcal{R}^{Coq}_{\Pi Dis}) \\
(\Sigma y : A.B)[x \leftarrow tt] & & \Sigma y : (\Pi x : unit.A).\Pi x : unit.B[y \leftarrow (y\ x)] \\
\equiv & & \downarrow (\mathcal{R}^{Coq}_{\Pi Unl}) + (\mathcal{R}^{Coq}_{\Sigma R}) \\
\Sigma y.A[x \leftarrow tt].B[x \leftarrow tt] & & \Sigma y : (\Pi x : unit.A).B[y \leftarrow (y\ x)][x \leftarrow tt] \\
 & & \equiv \\
 & & \Sigma y : (\Pi x : unit.A).B[y \leftarrow (y\ tt)]
\end{array}$$

Thus, to have canonical normal form, we defined a reduction strategy called $\mathcal{STR}^{Coq}$ and based on $\mathcal{R}^{Coq}$. This strategy respects the following partial order:

$$\mathcal{R}^{Coq}_{\Sigma Ass} =_{\mathcal{R}^{Coq}} \mathcal{R}^{Coq}_{\Sigma Cur} >_{\mathcal{R}^{Coq}} \mathcal{R}^{Coq}_{*U*} >_{\mathcal{R}^{Coq}} \mathcal{R}^{Coq}_{\Pi Dis}$$

This means that the rules $\mathcal{R}^{Coq}_{\Sigma Ass}$ and $\mathcal{R}^{Coq}_{\Sigma Cur}$ must be used equally before the rules $\mathcal{R}^{Coq}_{*U*}$ which in turn must be used equally before the rule $\mathcal{R}^{Coq}_{\Pi Dis}$.

We can show that $\mathcal{STR}^{Coq}$ is confluent and strongly normalizable. Due to the restriction, the normal forms are a little complicated and of the following form:

$$\Sigma \overrightarrow{x} : \overrightarrow{X}.T$$

Where there is no β-redex in $\{\overrightarrow{X}, T\}$ and:

- X_i is such that: $\Pi \overrightarrow{y_i} : \overrightarrow{Y_i}.\Sigma \overrightarrow{z_i} : \overrightarrow{Z_i}.U_i$ with $\overrightarrow{y_i} = \overrightarrow{0} \Rightarrow \overrightarrow{z_i} = \overrightarrow{0}$, $\overrightarrow{y_i} = \overrightarrow{0} \Rightarrow U_i \neq unit$ and where there is no Σ at the root of U_i;
- T is as follows: $T = \Pi \overrightarrow{v} : \overrightarrow{V}.W$ where there is no Σ at the root of V_i, W and where $V_i \neq unit$, $W \neq unit$.

Now, we can give the definition of the decision procedure:

Definition 2 (Decision procedure). *We call Dec^{Coq} the decision procedure which, for two types, $\mathcal{STR}^{Coq}$-normalizes them, then compares them modulo permutation of the Σ-components.*

We have some usual and expected properties:

[4] This mainly consists in orienting some rules of $\mathsf{Th}^{\mathsf{ECCE}}$ and ignoring the others.

Theorem 2 (Soundness and termination). *Dec^{Coq} is sound for* $\mathsf{Th}^{\mathsf{ECCE}}$ *and* *Dec^{Coq} terminates.*

Proof. The proof is quite trivial. The normalization uses the rules of $\mathcal{STR}^{Coq}$, that is to say, rules of $\mathcal{R}^{Coq}$, which are extracted from $\mathsf{Th}^{\mathsf{ECCE}}$. The comparison uses an axiom which can be deduced from $\mathsf{Th}^{\mathsf{ECCE}}$. For the termination, we know that $\mathcal{STR}^{Coq}$ is strongly normalizable and that permutations on a finite sequence are finite.

Of course, Dec^{Coq} is not complete for $\mathsf{Th}^{\mathsf{ECCE}}$ because we do not allow some rules to use "left-contextual" rules. However, Dec^{Coq} is a little more than the contextual part of $\mathsf{Th}^{\mathsf{ECCE}}$ because β-reduction can occur everywhere.

6 Implementation and examples

The implementation we carried out for Coq is called SearchIsos[5]. In fact, there are two tools: one inside the toplevel of Coq, which scans the current context, and another standalone tool, called Coq_SearchIsos, which scans the whole standard library of Coq.

In general, we suppose that users would use SearchIsos for tiny and trivial examples. Indeed, most of the time, users are interested in finding lemmas modulo α-conversion and permutation in Π-expressions. For instance, we may have such requests such as the following[6]:

```
Coq_SearchIsos < Time SearchIsos (A:Prop)A\/~A.
#Classical_Prop#--* [classic : (P:Prop)P\/~P]
Finished transaction in 1 secs (0.6u,0s)

Coq_SearchIsos < Time SearchIsos (b:bool)b=false->b=true->False.
#Bool#--* [eq_true_false_abs : (b:bool)b=true->b=false->False]
Finished transaction in 1 secs (0.716666666667u,0s)
```

As expected, possibilities about Σ-types are quite powerful. For example, we can hide the Archimedian axiom of the real numbers in an inductive type and, to find it again, we can use usual existential quantifiers, which are much more natural:

```
Coq < Require Reals.

Coq < Inductive Tarchi [r:R]:Set:=
Coq <     CTarchi:(n:nat)(gt n O)->(Rgt (INR n) r)->(Tarchi r).
Tarchi_ind is defined
Tarchi_rec is defined
Tarchi_rect is defined
Tarchi is defined
```

[5] See [4] for documentation.

[6] For all these tests, we used a PWS 500 Digital-Alpha station with bytecode.

```
Coq < Axiom archi:(r:R)(Tarchi r).
archi is assumed

Coq < Time SearchIsos (r:R){n:nat|(gt n O)/\(Rgt (INR n) r)}.
#--* [archi : (r:R)(Tarchi r)]
Finished transaction in 0 secs (0.266666666667u,0s)
```

As concerns *unit*, we must not forget that it includes all the inductive types with one empty constructor. So, the following result is not surprising:

```
Coq_SearchIsos < Time SearchIsos unit.
#Datatypes#<unit>#tt:unit
#Logic#<True>#I:True
#Logic_Type#<UnitT>#IT:UnitT
Finished transaction in 0 secs (0.516666666667u,0s)
```

In fact, we think that users are also very interested in congruences and the use of metavariables. We have not yet implemented these possibilities but we plan to do so soon.

7 Conclusion

7.1 Summary

In this work, we have achieved three goals:

- we have developed a theory $\mathsf{Th}^{\mathsf{ECCE}}$ with "ad hoc" contextual rules, which is sound for ECCE;
- we have made contextual restrictions on $\mathsf{Th}^{\mathsf{ECCE}}$ to build a decision procedure Dec^{Coq} which is sound for $\mathsf{Th}^{\mathsf{ECCE}}$ and which is an approximation of the contextual part of $\mathsf{Th}^{\mathsf{ECCE}}$;
- we have implemented Dec^{Coq} in a tool called SearchIsos.

7.2 Future work

Several aspects remain to be explored:

- subsitution to the left of an anonymous binder (when Π is a $\rightarrow$): this weakens the restrictions on the contextual rules and we capture types users may expect to capture;
- introduction of congruences: this possiblity seems to have priority. Indeed, for example, SearchIsos must deal with symmetry or associativity of some operators;
- pattern-matching: just like congruences, this must be implemented quite quickly. SearchIsos could then subsume the current command `Search` which uses a basic pattern-matching to find all the lemmas with a certain identifier as head-constant in their conclusion;

– inductive types: as we saw previously, this can only be a kind of α-conversion with respect to constructor names or an identification which is more semantic (but difficult to decide).

This kind of tool could also be useful for automated theorem proving, where the search of a given lemma would be done modulo type isomorphisms. In this perspective, invertible terms would have to be provided.

References

1. Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer-Verlag, 1996.
2. Maria-Virginia Aponte, Roberto Di Cosmo, and Catherine Dubois. Signature subtyping modulo type isomorphisms, 1998. http://www.pps.jussieu.fr/~dicosmo/ADCD97.ps.gz.
3. Andrew W. Appel et al. *Standard ML of New Jersey User's Guide*. New Jersey, 1998. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/index.html.
4. Bruno Barras et al. *The Coq Proof Assistant Reference Manual Version 6.3.1*. INRIA-Rocquencourt, May 2000. http://coq.inria.fr/doc-eng.html.
5. Dave Berry et al. *Edinburgh SML*. Laboratory for Foundations of Computer Science, University of Edinburgh, 1991. http://www.lfcs.informatics.ed.ac.uk/software/.
6. Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of type. In *Mathematical Structures in Computer Science*, volume 2(2), pages 231–247, 1992.
7. Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.
8. M. J. C. Gordon et al. A metalanguage for interactive proof in LCF. In *5th POPL*, ACM, 1978.
9. Simon Peyton Jones et al. *Haskell 98*, February 1999. http://www.haskell.org/definition/.
10. Thomas Kolbe and Christoph Walther. Adaptation of proofs for reuse. In *Adaptation of Knowledge for Reuse*, AAAI Fall Symposium, 1995. http://www.aic.nrl.navy.mil/~aha/aaai95-fss/papers.html#kolbe.
11. Thomas Kolbe and Christoph Walther. Proof management and retrieval. In *Working Notes of the IJCAI Workshop*, Formal Approaches to the Reuse of Plans, Proofs, and Programs, 1995. http://www.informatik.uni-freiburg.de/~koehler/ijcai-95/ijcai-ws/kolbe.ps.gz.
12. Xavier Leroy. *The Caml Light system, documentation and user's guide Release 0.74*. INRIA-Rocquencourt, December 1997. http://caml.inria.fr/man-caml/index.html.
13. Xavier Leroy et al. *The Objective Caml system release 3.00*. INRIA-Rocquencourt, April 2000. http://caml.inria.fr/ocaml/htmlman/.

14. Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, July 1990.
15. A. Mili, R. Mili, and R. T. Mittermeir. A survey of software reuse libraries. In *Annals of Software Engineering*, volume 5, pages 349–414, 1998.
16. Mikael Rittri. Using types as search keys in function libraries. In *Journal of Functional Programming*, volume 1(1), pages 171–89, 1991.
17. Sergei Soloviev. The category of finite sets and cartesian closed categories. In *Journal of Soviet Mathematics*, volume 22(3), pages 154–172, 1983.
18. Jérôme Vouillon and Julien Jalon. CamlSearch. Master's thesis, LIENS, 1994. http://caml.inria.fr/contribs-eng.html.

A Inference rules of ECCE

$$\frac{}{[]\text{ is well formed}} \quad (\mathsf{ECCE_{Cxt0}})$$

$$\frac{\Gamma \vdash A : Type_i \qquad x \notin FV(\Gamma)}{\Gamma, x : A_i \text{ is well formed}} \quad (\mathsf{ECCE_{Cxt1}})$$

$$\frac{x \in \Gamma \qquad \Gamma \text{ is well formed}}{\Gamma \vdash x : A} \quad (\mathsf{ECCE_{Var}})$$

$$\frac{\Gamma \text{ is well formed}}{\Gamma \vdash Prop : Type_0} \quad (\mathsf{ECCE_{Prop}})$$

$$\frac{\Gamma \text{ is well formed}}{\Gamma \vdash Type_i : Type_{i+1}} \quad (\mathsf{ECCE_{Univ}})$$

$$\frac{\Gamma \text{ is well formed}}{\Gamma \vdash unit : Prop} \quad (\mathsf{ECCE_{Unit}})$$

$$\frac{\Gamma \text{ is well formed}}{\Gamma \vdash () : unit} \quad (\mathsf{ECCE_{()}})$$

$$\frac{\Gamma, x : A \vdash P : Prop}{\Gamma \vdash \Pi x : A.P : Prop} \quad (\mathsf{ECCE_{Prod0}})$$

$$\frac{\Gamma \vdash A : Type_i \qquad \Gamma, x : A \vdash B : Type_i}{\Gamma \vdash \Pi x : A.B : Type_i} \quad (\mathsf{ECCE_{Prod1}})$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B} \quad (\mathsf{ECCE_{Lam}})$$

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[x \leftarrow N]} \quad (\mathsf{ECCE_{App}})$$

$$\frac{\Gamma \vdash A : Type_i \qquad \Gamma, x : A \vdash B : Type_i}{\Gamma \vdash \Sigma x : A.B : Type_i} \quad (\mathsf{ECCE_{Sig}})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma N : B[x \leftarrow M] \qquad \Gamma, x : A \vdash B : Type_i}{\Gamma \vdash (M, N)_{\Sigma x : A.B} : \Sigma x : A.B} \quad (\mathsf{ECCE_{Pair}})$$

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash (pi_1\ M) : A} \quad (\mathsf{ECCE_{Proj0}})$$

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash (pi_2\ M) : B} \quad (\mathsf{ECCE_{Proj1}})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A' : Type_i \qquad A \simeq A'}{\Gamma \vdash M : A'} \quad (\mathsf{ECCE_{Conv}})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A' : Type_i \qquad A \prec A'}{\Gamma \vdash M : A'} \quad (\mathsf{ECCE_{Cum}})$$

B Axioms and inference rules of $\mathsf{Th}^{\mathsf{ECCE}}$

$$\underset{\left(\begin{smallmatrix} \lambda x : A.x \\ \lambda x : B.x \end{smallmatrix}\right)}{A = B} \text{ if } A \simeq B \ (\mathsf{Th}^{\mathsf{ECCE}}_{\beta})$$

$$\underset{\left(\begin{smallmatrix} \lambda c : (\Sigma x : unit.A).(\pi_2\ c) \\ \lambda a : A[x \leftarrow tt].(tt,\ a) \end{smallmatrix}\right)}{\Sigma x : unit.A = A[x \leftarrow tt]} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{UL}})$$

$$\underset{\left(\begin{smallmatrix} \lambda c : (\Sigma x : A.B).(\pi_2\ c,\ \pi_1\ c) \\ \lambda c : (\Sigma x : B.A).(\pi_2\ c,\ \pi_1\ c) \end{smallmatrix}\right)}{\Sigma x : A.B = \Sigma x : B.A} \text{ if } x \notin FV(A, B) \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{Com}})$$

$$\underset{\left(\begin{smallmatrix} \lambda c : (\Sigma x : unit.A).(\pi_2\ c) \\ \lambda a : A[x \leftarrow tt].(tt,\ a) \end{smallmatrix}\right)}{\Sigma x : unit.A = A[x \leftarrow tt]} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{UL}})$$

$$\underset{\left(\begin{smallmatrix} \lambda z : (\Sigma x : (\Sigma y : A.B).C).(\pi_1\ (\pi_1\ z), (\pi_2\ (\pi_1\ z), \pi_2\ z)) \\ \lambda z : (\Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x,\ y)]).((\pi_1\ z, \pi_1\ (\pi_2\ z)), \pi_2\ (\pi_2\ z)) \end{smallmatrix}\right)}{\Sigma x : (\Sigma y : A.B).C = \Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)]} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{Ass}})$$

$$\underset{\left(\begin{smallmatrix} \lambda f : (\Pi x : A.unit).tt \\ \lambda u : unit.\lambda x : A.tt \end{smallmatrix}\right)}{\Pi x : A.unit = unit} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{UR}})$$

$$\underset{\left(\begin{smallmatrix} \lambda f : (\Pi x : (\Sigma y : A.B).C).\lambda x : A.\lambda y : B[y \leftarrow x].f\ (x,\ y) \\ \lambda f : (\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x,\ y)]).\Pi x : (\Sigma y : A.B).f\ (\pi_1\ x)\ (\pi_2\ x) \end{smallmatrix}\right)}{\Pi x : (\Sigma y : A.B).C = \Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{Cur}})$$

$$\underset{\left(\begin{smallmatrix} \lambda f : (\Pi x : unit.A).f\ tt \\ \lambda a : A[x \leftarrow tt].\lambda x : unit.a \end{smallmatrix}\right)}{\Pi x : unit.A = A[x \leftarrow tt]} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{UL}})$$

$$\underset{\left(\begin{smallmatrix} \lambda f : (\Pi x : A.\Sigma y : B.C).(\Pi x : A.(\pi_1\ (f\ x)), \Pi x : A.(\pi_2\ (f\ x))) \\ \lambda c : (\Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y\ x)]).\lambda x : A.((\pi_1\ c)\ x, (\pi_2\ c)\ x) \end{smallmatrix}\right)}{\Pi x : A.\Sigma y : B.C = \Sigma y : (\Pi x : A.B).\Pi x : A.C[y \leftarrow (y\ x)]} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{Dis}})$$

$$\frac{}{\underset{\left(\begin{smallmatrix} \lambda x : A.x \\ \lambda x : A.x \end{smallmatrix}\right)}{A = A}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\mathsf{Ref}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = A'}}{\underset{\left(\begin{smallmatrix} \lambda f : (\Pi x : A.B).\lambda x : A'.f\ (\tau\ x) \\ \lambda f : (\Pi x : A'.B[x \leftarrow (\tau\ x)]).\lambda x : A.f\ (\sigma\ x) \end{smallmatrix}\right)}{\Pi x : A.B = \Pi x : A'.B[x \leftarrow (\tau\ x)]}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{L}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = B}}{\underset{\left(\begin{smallmatrix} \tau \\ \sigma \end{smallmatrix}\right)}{B = A}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\mathsf{Sym}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = A'}}{\underset{\left(\begin{smallmatrix} \lambda c : (\Sigma x : A.B).(\sigma\ (\pi_1\ c),\ \pi_2\ c) \\ \lambda c : (\Sigma x : A'.B[x \leftarrow (\tau\ x)]).(\tau\ (\pi_1\ c),\ \pi_2\ c) \end{smallmatrix}\right)}{\Sigma x : A.B = \Sigma x : A'.B[x \leftarrow (\tau\ x)]}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{L}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = B} \quad \underset{\left(\begin{smallmatrix} \sigma' \\ \tau' \end{smallmatrix}\right)}{B = C}}{\underset{\left(\begin{smallmatrix} \sigma' \circ \sigma \\ \tau \circ \tau' \end{smallmatrix}\right)}{A = C}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\mathsf{Trs}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = A'}}{\underset{\left(\begin{smallmatrix} \lambda f : (\Pi x : B.A).\lambda x : B.\sigma\ (f\ x) \\ \lambda f : (\Pi x : B.A').\lambda x : B.\tau\ (f\ x) \end{smallmatrix}\right)}{\Pi x : B.A = \Pi x : B.A'}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Pi\mathsf{R}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = B}}{\underset{\left(\begin{smallmatrix} \rho\sigma \\ \rho\tau \end{smallmatrix}\right)}{\rho A = \rho B}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\mathsf{Sbs}})$$

$$\frac{\underset{\left(\begin{smallmatrix} \sigma \\ \tau \end{smallmatrix}\right)}{A = A'}}{\underset{\left(\begin{smallmatrix} \lambda c : (\Sigma x : B.A).(\pi_1\ c, \sigma\ (\pi_2\ c)) \\ \lambda c : (\Sigma x : B.A').(\pi_1\ c, \tau\ (\pi_2\ c)) \end{smallmatrix}\right)}{\Sigma x : B.A = \Sigma x : B.A'}} \ (\mathsf{Th}^{\mathsf{ECCE}}_{\Sigma\mathsf{R}})$$

C Rules of $\mathcal{R}^{Coq}$

$A \xrightarrow{\mathcal{R}^{Coq}}$
B if $A \xrightarrow{\beta} B$ $(\mathcal{R}^{Coq}_{\beta})$

$\Sigma x : A.unit \xrightarrow{\mathcal{R}^{Coq}}$
A $(\mathcal{R}^{Coq}_{\Sigma\mathsf{UR}})$

$\Sigma x : (\Sigma y : A.B).C \xrightarrow{\mathcal{R}^{Coq}}$
$\Sigma x : A.\Sigma y : B[y \leftarrow x].C[x \leftarrow (x, y)]$ $(\mathcal{R}^{Coq}_{\Sigma\mathsf{Ass}})$

$\Sigma x : unit.A \xrightarrow{\mathcal{R}^{Coq}}$
$A[x \leftarrow tt]$ $(\mathcal{R}^{Coq}_{\Sigma\mathsf{UL}})$

$\Pi x : (\Sigma y : A.B).C \xrightarrow{\mathcal{R}^{Coq}}$
$\Pi x : A.\Pi y : B[y \leftarrow x].C[x \leftarrow (x, y)]$ $(\mathcal{R}^{Coq}_{\Sigma\mathsf{Cur}})$

$\Pi x : A.unit \xrightarrow{\mathcal{R}^{Coq}}$
$unit$ $(\mathcal{R}^{Coq}_{\Pi\mathsf{UR}})$

$\Pi x : A.\Sigma y : B. \xrightarrow{\mathcal{R}^{Coq}} C$
$\Sigma y : (\Pi x : A.B).(\Pi x : A.C[y \leftarrow (y\ x)]$ $(\mathcal{R}^{Coq}_{\Pi\mathsf{Dis}})$

$\Pi x : unit.A \xrightarrow{\mathcal{R}^{Coq}}$
$A[x \leftarrow tt]$ $(\mathcal{R}^{Coq}_{\Pi\mathsf{UL}})$

$$\frac{A \xrightarrow{\mathcal{R}^{Coq}} A'}{\rho A \xrightarrow{\mathcal{R}^{Coq}} \rho A'}\ \mathcal{R}^{Coq}_{\mathsf{Sbs}}$$

$$\frac{A \xrightarrow{\mathcal{R}^{Coq}_{\beta}} A'}{\Pi x : A.B \xrightarrow{\mathcal{R}^{Coq}} \Pi x : A'.B}\ \mathcal{R}^{Coq}_{\beta\Pi\mathsf{L}}$$

$$\frac{A \xrightarrow{\mathcal{R}^{Coq}} A'}{\Pi x : B.A \xrightarrow{\mathcal{R}^{Coq}} \Pi x : B.A'}\ \mathcal{R}^{Coq}_{\Pi\mathsf{R}}$$

$$\frac{A \xrightarrow{\mathcal{R}^{Coq}_{\beta}} A'}{\Sigma x : A.B \xrightarrow{\mathcal{R}^{Coq}} \Sigma x : A'.B}\ \mathcal{R}^{Coq}_{\beta\Sigma\mathsf{L}}$$

$$\frac{A \xrightarrow{\mathcal{R}^{Coq}} A'}{\Sigma x : B.A \xrightarrow{\mathcal{R}^{Coq}} \Sigma x : B.A'}\ \mathcal{R}^{Coq}_{\Sigma\mathsf{R}}$$