

# Subset coercions in COQ

Matthieu Sozeau

LRI, Paris Sud XI University  
sozeau@lri.fr

**Abstract.** We propose a new language for writing programs with dependent types which can be elaborated into partial COQ terms. This language permits to establish a phase distinction between writing and proving algorithms in the COQ environment. Concretely, this means allowing to write algorithms as easily as in a practical functional programming language whilst giving them as rich a specification as desired and proving that the code meets the specification using the whole COQ proof apparatus. This is achieved by extending conversion to an equivalence which relates types and subsets based on them, a technique originating from the “*Predicate subtyping*” feature of PVS and following mathematical convention. The typing judgements can be translated to the Calculus of (Co-)Inductive Constructions (CIC) by means of an interpretation which inserts coercions at the appropriate places. These coercions can contain existential variables representing the propositional parts of the final term, corresponding to proof obligations (or PVS type-checking conditions). A prototype implementation of this process is integrated with the COQ environment.

## 1 Introduction

There are many means to program in the COQ environment [1]. One can write programs as in ML and prove properties about them separately, loosing the possibility of using dependent types in specifications, or give a rich type expressing them as a goal and use the proof tactics to solve it, producing a corresponding program by the Curry-Howard isomorphism but having much less control on its algorithmic essence. It is however difficult to mix the two methods (writing code and proving interactively) using a rich specification. Indeed, when using simple terms and types (ML) or even complex terms and types (COQ), we can have decidable type-checking. However, when using simple terms to represent inhabitants of complex types, we immediately get undecidability of type-checking, as the terms do not give enough information. Consider for example the function `tl` which returns the tail of a non-empty list. In ML:

```
let tl = function hd :: t1 -> t1 | [] -> assert(false)
```

This function is partial, its domain is reduced to non-empty lists. In COQ, we would rather write the following:

```

Definition t1 (l : list A) : option (list A) :=
  match l with
  | hd :: tl => Some tl
  | [] => None
  end.

```

The option type is the usual way to encode partiality in Coq. However we would more naturally constrain the argument `l` to non-empty lists to be more faithful to the original ML code, and also attach a property to the result to express more precisely what `t1` does:

```

Definition t1 (l : list A) : (l <> []) ->
  { l' : list A | exists a, a :: l' = l }.

```

Now the definition's body requires some heavy plumbing of the code which is not affordable when programming. Moreover, the user is forced to give a proof term justifying that `l <> []` when calling the function. We propose a solution to overcome these difficulties, based on the *Predicate subtyping* [2] technique of PVS [3]. It is separated into two phases. First, we have a weak, *decidable* type-checking procedure which does not require proofs to be present in the code when constructing objects of a subset type. In our new language RUSSELL, the following is a well-typed term:

```

Program Definition t1 ( l : list A | l <> [] ) :
  { l' : list A | exists a, a :: l' = l } :=
  match l with
  | hd :: tl -> tl
  | [] -> []
  end

```

The specification shows that we are defining a partial function and enforces a relation between input and output, using a dependent type, yet the code remains as simple as the ML definition. This is only possible because we do not require the user to write proofs in the code. After type-checking, there is an automatic elaboration into partial Coq terms, which collects obligations the user has to prove. In our example, the proof assistant will ask the user to prove that:

1. The list `tl` has the property `exists a, a :: tl = hd :: tl`, and
2. In the context where `l` is a non-empty list and `l = []`, the list `tl` has the property `exists a, a :: [] = []` (which should be obvious as the context is contradictory).

This solution also provides facilities to express properties with a more mathematical flavour using subsets, bridging a gap between mathematical convention and type theory.

The PROGRAM tactic by C. Parent [4] had the same goal as ours but a slightly different method for achieving it. It was strongly linked to the extraction mechanism included in Coq, both theoretically and practically. Sketching the mechanism, she defined a weakened extraction operation on CIC terms which could

be inverted because it left enough information in the extracted term to rebuild a partial proof. The mechanism, while general and theoretically well thought out, required some heuristics and did not integrate smoothly with the COQ environment. In particular it lacked the pervasiveness our method has, being applicable in a wide variety of situations in the proof assistant environment.

Instead of trying to find a general method for synthesizing proofs from programs in the Calculus of (Co-)Inductive Constructions, we have integrated a method which permits to link strong specifications and purely algorithmical code. This method, known as *Predicate subtyping* in the PVS system, has been used with great success and fits naturally with how we write specifications using the subset type [5]. The main contribution of this paper is to show how the *Predicate subtyping* method can be adapted in a proof assistant with proof terms, which formally justifies the extension in the first place.

The remaining of the article is organised as follows: in section 2 we present a type system based on CIC, which integrates subset “subtyping”, and prove decidability of type-checking. Then, we show how it relates to CIC by means of translations between judgements of the two systems in section 3. Next we present a prototype implementation in section 4, and finally we give our conclusions on this work and discuss future directions in section 5.

## 2 RUSSELL

The name of our language is an homage to the mathematician Bertrand Russell who discovered the famous paradox of set theory with the unrestricted comprehension axiom. In this theory, it is possible to construct the set  $X = \{x \mid x \notin x\}$  whose definition is circular. Clearly, if  $X \in X$  then  $X \notin X$  and if  $X \notin X$  then  $X \in X$ , hence we have an inconsistency. RUSSELL was one of the pioneers of type theory when he devised a set theory with a restricted comprehension axiom that permitted to create subsets only from already defined subsets, forbidding the definition of  $X$ .

In COQ, the distinction between informative and propositional parts of a term is formalised by the **Prop/Set** sorts. In RUSSELL, we have special support for propositions appearing in subset types. By delimiting the use of propositions, we can separate code from proof.

Informally, this idea is already present in mathematics. When you have an element of subset  $\{x \in S \mid P\}$ , you can freely forget about the property  $P$  and use any operation which is defined on  $S$ . Conversely, when you want to use an operation defined on a subset, say  $f : \{x : \mathbb{N} \mid P\} \rightarrow X$ , you usually prove *first* that you apply it only to elements having the desired property. For example  $(f\ 2)$  is a correct application only if  $(P\ 2)$  is provable. In the context of formal development of programs, such a workflow is not entirely satisfactory because it forces one to create objects and prove properties about them *at the same time*. In COQ, we are forced to apply  $f$  to an object  $(\text{elt nat } P\ 2\ p)$  where  $p$  is a proof of  $(P\ 2)$  (Figure 1 presents the definition of the subset type in COQ). We would like to be able to prove that our usage of partial functions is correct only

after the program is written. Subsets are particularly well-suited in this respect because they separate the objects we want to manipulate and their associated properties.

$$\begin{array}{c}
\text{SUBSET} \frac{\Gamma \vdash A : \mathbf{Set} \quad \Gamma, x : A \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : A \mid P \} : \mathbf{Set}} \\
\text{ELEMENT} \frac{\Gamma \vdash a : A \quad \Gamma \vdash p : P[a/x] \quad \Gamma \vdash \{ x : A \mid P \} : \mathbf{Set}}{\Gamma \vdash \text{elt } A (\lambda x : A. P) a p : \{ x : A \mid P \}} \\
\text{SUBSET-}\sigma_1 \frac{\Gamma \vdash t : \{ x : A \mid P \}}{\Gamma \vdash \sigma_1 t : A} \quad \text{SUBSET-}\sigma_2 \frac{\Gamma \vdash t : \{ x : A \mid P \}}{\Gamma \vdash \sigma_2 t : P[\sigma_1 t/x]}
\end{array}$$

**Fig. 1.** Subset type in CIC

## 2.1 From *Predicate subtyping* to subset equivalence

The *Predicate subtyping* mechanism [2] is an extension of the PVS type system which internalises this idea. Concretely, this means that the following rules are derivable in PVS:

$$\frac{\Gamma \vdash t : \{ x : T \mid P[x] \}}{\Gamma \vdash t : T} \quad \frac{\Gamma \vdash t : T \quad \vdash_{\Gamma} P[t]}{\Gamma \vdash t : \{ x : T \mid P[x] \}}$$

The first one formalises the fact that an object of a subset based on  $T$  is an object of type  $T$ . The second one permits using an object of type  $T$  as an object of type  $\{ x : T \mid P[x] \}$ , but it generates a *type-checking condition*  $\vdash_{\Gamma} P[t]$  which will need to be discharged later. Effectively, the typing algorithm of PVS collects the conditions that must be satisfied for the term to be accepted as a valid definition. However, the acceptance criteria can be rather large in PVS. By design, when proving in PVS, the trusted code base (TCB hereafter) is the entire system, not only the typing system but also the various decision procedures and tactics used to build proofs automatically or interactively. It is nonetheless a widely used proof assistant and the predicate subtyping feature has apparently helped to build a consequent library of certified code. On the other hand, COQ has a small TCB and greater expressiveness but less automation and methodology to build certified programs. We capitalise on the PVS success to make COQ more usable for this kind of tasks, and as we will see, it will have other benefits.

## 2.2 A weaker type system

To formalise this idea in COQ, we simply weaken the type system so that it doesn't require the terms to contain the proof components for objects of subset types. This permits to have a simple language for code while retaining the richness of COQ's specification language. Once we have a derivation in this new type system, we can translate it to a partial COQ derivation, where the missing

parts are represented by metavariables. It can then be completed by instantiating these holes with actual proofs.

$$\begin{array}{c}
\text{WF-EMPTY} \frac{}{\vdash \square \mathbf{wf}} \quad \text{WF-VAR} \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A \mathbf{wf}} \quad s \in \mathcal{S} \wedge x \notin \Gamma \\
\text{VAR} \frac{\vdash \Gamma \mathbf{wf} \quad x : A \in \Gamma}{\Gamma \vdash x : A} \quad \text{AXIOM} \frac{\vdash \Gamma \mathbf{wf}}{\Gamma \vdash s_1 : s_2} \quad (s_1, s_2) \in \mathcal{A} \\
\text{PROD} \frac{\Gamma \vdash T : s_1 \quad \Gamma, x : T \vdash U : s_2}{\Gamma \vdash \Pi x : T.U : s_2} \\
\text{ABS} \frac{\Gamma \vdash \Pi x : T.U : s \quad \Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : \Pi x : T.U} \quad \text{APP} \frac{\Gamma \vdash f : \Pi x : V.W \quad \Gamma \vdash u : V}{\Gamma \vdash (fu) : W[u/x]} \\
\text{SUM} \frac{\Gamma \vdash T : s \quad \Gamma, x : T \vdash U : s}{\Gamma \vdash \Sigma x : T.U : s} \quad s \in \{\mathbf{Prop}, \mathbf{Set}\} \\
\text{PAIR} \frac{\Gamma \vdash \Sigma x : T.U : s \quad \Gamma \vdash t : T \quad \Gamma \vdash u : U[t/x]}{\Gamma \vdash (t, u)_{\Sigma x : T.U} : \Sigma x : T.U} \\
\text{PI-1} \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_1 t : T} \quad \text{PI-2} \frac{\Gamma \vdash t : \Sigma x : T.U}{\Gamma \vdash \pi_2 t : U[\pi_1 t/x]} \\
\text{CONV} \frac{\Gamma \vdash t : U \quad \Gamma \vdash U \equiv_{\beta\pi} T : s}{\Gamma \vdash t : T}
\end{array}$$

**Fig. 2.** CIC typing judgement

**Type system** RUSSELL's type system is based on the Calculus of Inductive Constructions (figure 2) [6], with sigma types but without universes. This restriction may be removed in future work, but causes no problem for our main purpose which is programming. We omit inductive constructs here, as they can be considered as constants and leave their treatment as future work. The judgement  $\Gamma \vdash t : T$  means  $t$  is a well-typed term of type  $T$  in environment  $\Gamma$ .

Following the presentation as a Pure Type System, the set of sorts  $\mathcal{S}$  is defined as  $\{\mathbf{Set}, \mathbf{Prop}, \mathbf{Type}\}$ . As usual, we let  $s, s_i$  for  $i \in \mathbb{N}$  range over sorts. The axioms are  $\mathcal{A} = \{(\mathbf{Set}, \mathbf{Type}), (\mathbf{Prop}, \mathbf{Type})\}$  and the set of rules  $\mathcal{R}$  is defined by the functional relation  $\forall s_1 s_2, (s_1, s_2, s_2) \in \mathcal{R}$ . We allow products  $\Pi x : A, B$  where  $A : \mathbf{Prop}$  and  $B : \mathbf{Set}$  but the user is encouraged to encode them as  $\Pi x : \{- : \mathit{unit} \mid A\}, B$  to benefit from the subtyping mechanism which we will define later.

The reduction rules of this system are the standard  $\beta$  and  $\pi$  for projections. We denote by  $\downarrow x$  the complete normal form of  $x$  and by  $x^\downarrow$  its head normal form. We use a standard judgemental equality  $\Gamma \vdash T \equiv_{\beta\pi} U : s$ .

We allow to form dependent sums  $\Sigma x : U.V$  (SUM, SUM-DEP, PI-1, PI-2 rules) only when  $U, V : \mathbf{Set}$  or  $U, V : \mathbf{Prop}$ . The first sum represents dependent pairs, useful when defining functions returning a tuple, the latter represent dependent pairs of propositions, which is most often used as conjunction of

$$\text{COERCE} \frac{\Gamma \vdash t : U \quad \Gamma \vdash U \triangleright T : s}{\Gamma \vdash t : T}$$

$$\text{SUBSET} \frac{\Gamma \vdash U : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} : \mathbf{Set}}$$

**Fig. 3.** RUSSELL new rules

propositions. The dependent pair with  $U : \mathbf{Set}$  and  $V : \mathbf{Prop}$  is the subset type which we distinguish (rule SUBSET). The last possible pair  $U : \mathbf{Prop}, V : \mathbf{Set}$  is forbidden because it corresponds to a pair where the last component, which is informative, may depend on a particular proof of the proposition  $U$ . This is quite contrary to the mantra “Computations do not depend on proofs” which governs our programming language. If there is no dependence then  $U$  and  $V$  can be swapped.

In RUSSELL, the conversion rule CONV is replaced by a new subsumption rule COERCE (figure 3) which will implement the subset equivalence. The judgement  $\Gamma \vdash T \triangleright U : s$  means  $T$  is equivalent to  $U$  in environment  $\Gamma$ , both sorted with  $s$ . The essence of our equivalence is to identify subset types if they have equivalent supports, hence we have the following property:

**Proposition 1 (Subset erasure and consistency).** *If we erase subset types from RUSSELL terms and rules, leaving only the supports, we get a valid CIC term and derivation, hence  $\perp$  is not provable in RUSSELL.*

*Proof.* By erasing subsets in RUSSELL rules, the SUBSET rule becomes admissible and the COERCE rule becomes CONV. By eliminating the subset type in a term we get a valid CIC term. Hence to each RUSSELL derivation corresponds a CIC derivation.

**Equivalence** We have renamed the technique from *Predicate subtyping* to *subset equivalence* because we have a symmetric relation, contrary to usual subtyping relations. It also conveys the idea that it can include the usual  $\beta\pi$ -conversion directly in the judgement. The judgement  $\Gamma \vdash T \triangleright U : s$  (figure 4 on the facing page) reads  $T$  is equivalent to  $U$  in environment  $\Gamma$ , both being sorted by  $s$ .

The rule  $\triangleright$ -CONV integrates  $\beta\pi$ -conversion in the judgement. We use a judgemental equality here, which will be refined by the usual conversion relation later. The  $\triangleright$ -SYM and  $\triangleright$ -TRANS rules ensure that our judgement builds an *equivalence* and has proper modularity. The next two rules ( $\triangleright$ -PROD and  $\triangleright$ -SUM) do context closure for dependent products and sums. It is remarkable that we use contravariance for domains in the  $\triangleright$ -PROD rule, not restricting to invariance as in PVS. It is accessory here, as we could have used *covariance* and still get the same judgements because we have symmetry. However, it will become important when we create coercions (see figure 8 on page 11).

The really interesting rules are  $\triangleright$ -SUBSET and  $\triangleright$ -PROOF. The first one allows to use an object of a subset type as an object of its support type. The later allows (maybe abusively) to consider an object of any type as an object of any subset

$$\begin{array}{c}
\triangleright\text{-CONV} \frac{\Gamma \vdash T \equiv_{\beta\pi} U : s}{\Gamma \vdash T \triangleright U : s} \\
\triangleright\text{-SYM} \frac{\Gamma \vdash U \triangleright T : s}{\Gamma \vdash T \triangleright U : s} \quad \triangleright\text{-TRANS} \frac{\Gamma \vdash S \triangleright T : s \quad \Gamma \vdash T \triangleright U : s}{\Gamma \vdash S \triangleright U : s} \\
\triangleright\text{-PROD} \frac{\Gamma \vdash U \triangleright T : s_1 \quad \Gamma, x : U \vdash V \triangleright W : s_2}{\Gamma \vdash \Pi x : T.V \triangleright \Pi x : U.W : s_2} \\
\triangleright\text{-SUM} \frac{\Gamma \vdash T \triangleright U : s \quad \Gamma, x : T \vdash V \triangleright W : s}{\Gamma \vdash \Sigma x : T.V \triangleright \Sigma y : U.W : s} \quad s \in \{\mathbf{Set}, \mathbf{Prop}\} \\
\triangleright\text{-SUBSET} \frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : U \vdash P : \mathbf{Prop}}{\Gamma \vdash \{ x : U \mid P \} \triangleright V : \mathbf{Set}} \\
\triangleright\text{-PROOF} \frac{\Gamma \vdash U \triangleright V : \mathbf{Set} \quad \Gamma, x : V \vdash P : \mathbf{Prop}}{\Gamma \vdash U \triangleright \{ x : V \mid P \} : \mathbf{Set}}
\end{array}$$

**Fig. 4.** RUSSELL conversion

based on this type. We must check that the property  $P$  is well-formed, but we do not care about its provability.

**Properties** We have proved the metatheory for this system in COQ (<http://www.lri.fr/~sozeau/research/russell/proof.en.html>). We have assumed that it is strongly normalising (SN) but we have proved subject reduction (SR) for it: if  $\Gamma \vdash t : T$  and  $t \rightarrow_{\beta\pi} t'$  then  $\Gamma \vdash t' : T$  (our proof of SR does not depend on SN). Gang Chen [7] has studied various type systems from the  $\lambda$ -cube extended by subtyping or coercive subtyping [8], including the Calculus of Constructions, and proved such results as SN and SR for them. It requires a very careful analysis of the system to avoid cycles in the proof due to the presence of dependent types and conversion. However we preferred to adapt the method of Robin Adams [9] to prove SR, because Gang Chen's method seemed very tied to the peculiarities of the system he studied. Strong normalisation could also be proved, perhaps more easily, using the generic proof method developed by Geuver [10], by finding a proper interpretation of RUSSELL terms into the Calculus of Constructions.

So, apart from strong normalisation, we have shown all the usual structural and metatheoretic properties of a dependent programming language, like weakening, thinning and substitution, stability by context coercion, etc. We stop here on the subject of theoretical properties of this first system, as it gives no new insights for our purpose and we will focus more on the algorithmic system's properties. Besides, the details of the formalisation and the proof fall outside the scope of this paper.

## Algorithmic typing

$$\begin{array}{c}
\text{APP} \frac{\Gamma \vdash_{\bullet} f : T \quad \mu_{\bullet}(T) = \Pi x : V.W \quad \Gamma \vdash_{\bullet} u : U \quad \Gamma \vdash_{\bullet} U \triangleright_{\bullet} V : s'}{\Gamma \vdash_{\bullet} (fu) : W[u/x]} \\
\text{PAIR} \frac{\Gamma \vdash_{\bullet} t : T' \quad \Gamma \vdash_{\bullet} T' \triangleright_{\bullet} T : s \quad \Gamma \vdash_{\bullet} \Sigma x : T.U : s \quad \Gamma \vdash_{\bullet} u : U' \quad \Gamma \vdash_{\bullet} U' \triangleright_{\bullet} U[t/x] : s}{\Gamma \vdash_{\bullet} (t, u)_{\Sigma x : T.U} : \Sigma x : T.U} \\
\text{PI-1} \frac{\Gamma \vdash_{\bullet} t : S \quad \mu_{\bullet}(S) = \Sigma x : T.U}{\Gamma \vdash_{\bullet} \pi_1 t : T} \quad \text{PI-2} \frac{\Gamma \vdash_{\bullet} t : S \quad \mu_{\bullet}(S) = \Sigma x : T.U}{\Gamma \vdash_{\bullet} \pi_2 t : U[\pi_1 t/x]}
\end{array}$$

**Fig. 5.** RUSSELL algorithmic typing, new rules

We use a standard transformation to get an algorithm from our typing rules, integrating the conversion (or subsumption) rule (here COERCE) to the premises of the other rules. Our algorithmic judgement  $\Gamma \vdash_{\bullet} t : T$  (figure 5) reads:  $t$  has type  $T$  in environment  $\Gamma$ . We need to introduce the notion of support for subset types in order to define our algorithmic system. Indeed, when typing an application, we need to ensure that the object we apply can be seen as an object of a product type modulo the equivalence.

$$\begin{array}{l}
\mu_{\bullet}(x) \Rightarrow \mu_{\bullet}(U) \text{ if } x^{\perp} = \{ x : U \mid P \} \\
\mu_{\bullet}(x) \Rightarrow x \quad \text{otherwise}
\end{array}$$

**Fig. 6.**  $\mu_{\bullet}()$  definition

$$\begin{array}{c}
\triangleright\text{-CONV} \frac{T \equiv_{\beta\pi} U \quad \Gamma \vdash_{\bullet} T, U : s}{\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s} \quad T = T^{\perp} \wedge T \neq \Pi, \Sigma, \{\} \wedge U = U^{\perp} \\
\triangleright\text{-}\downarrow \frac{\Gamma \vdash_{\bullet} T^{\perp} \triangleright_{\bullet} U^{\perp} : s \quad \Gamma \vdash_{\bullet} T, U : s}{\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s} \quad T \neq T^{\perp} \vee U \neq U^{\perp} \\
\triangleright\text{-PROD} \frac{\Gamma \vdash_{\bullet} U \triangleright_{\bullet} T : s_1 \quad \Gamma, x : U \vdash_{\bullet} V \triangleright_{\bullet} W : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Gamma \vdash_{\bullet} \Pi x : T.V \triangleright_{\bullet} \Pi x : U.W : s_2} \\
\triangleright\text{-SUM} \frac{\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s \quad \Gamma, x : T \vdash_{\bullet} V \triangleright_{\bullet} W : s}{\Gamma \vdash_{\bullet} \Sigma x : T.V \triangleright_{\bullet} \Sigma x : U.W : s} \quad s \in \{\mathbf{Set}, \mathbf{Prop}\} \\
\triangleright\text{-PROOF} \frac{\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : \mathbf{Set}}{\Gamma \vdash_{\bullet} T \triangleright_{\bullet} \{ x : U \mid P \} : \mathbf{Set}} \quad T = T^{\perp} \\
\triangleright\text{-SUBSET} \frac{\Gamma \vdash_{\bullet} U \triangleright_{\bullet} T : \mathbf{Set}}{\Gamma \vdash_{\bullet} \{ x : U \mid P \} \triangleright_{\bullet} T : \mathbf{Set}}
\end{array}$$

**Fig. 7.** RUSSELL algorithmic equivalence

The next thing to do is to construct a decidable judgement for equivalence given two types. We denote it by  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s$  (figure 7). In our case, we have to check

that conversion is decidable, which is the case as we can replace judgemental equality with the usual untyped convertibility  $\equiv_{\beta\pi}$  (we proved  $\Gamma \vdash T \equiv_{\beta\pi} U : s \Leftrightarrow T \equiv_{\beta\pi} U \wedge \Gamma \vdash T, U : s$ ). We also need to eliminate the transitivity and symmetry rules which are not syntax-directed. That is the purpose of the following theorem:

**Theorem 1 (Admissibility of transitivity and symmetry).**

1. If  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s$  and  $\Gamma \vdash_{\bullet} U \triangleright_{\bullet} V : s$ , then  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} V : s$ .
2. If  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s$  then  $\Gamma \vdash_{\bullet} U \triangleright_{\bullet} T : s$ .

*Proof.* 1. By induction on the sum of the depths of the two derivations.  
2. By induction, using a thinning lemma.

As usual, we have the following correspondence between the two systems:

**Theorem 2 (Soundness).** If  $\Gamma \vdash_{\bullet} t : T$  then  $\Gamma \vdash t : T$ . If  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s$  then  $\Gamma \vdash T \triangleright U : s$ .

**Theorem 3 (Completeness).** If  $\Gamma \vdash t : T$  then there exists  $T', s$  so that  $\Gamma \vdash_{\bullet} t : T'$  and  $\Gamma \vdash_{\bullet} T' \triangleright_{\bullet} T : s$ . If  $\Gamma \vdash T \triangleright U : s$  then  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s$ .

Finally, we can state the desired property of the algorithmic system:

**Theorem 4 (Decidability of typing).**  $\Gamma \vdash_{\bullet} t : T$  and  $\Gamma \vdash_{\bullet} T \triangleright_{\bullet} U : s$  are decidable problems.

*Proof.* Rules are syntax-directed.

We denote by  $\mathbf{type}_{\Gamma}(t)$  the function which returns the type of term  $t$  in context  $\Gamma$  such that  $\Gamma \vdash_{\bullet} t : \mathbf{type}_{\Gamma}(t)$ .

### 2.3 From CIC to RUSSELL

We have presented a calculus based on CIC with a stronger equivalence but with restricted typing rules. We will now see how they relate formally. We can build a forgetful map from terms of CIC to RUSSELL (interpreting inductives as constants). It bears relation to the  $\epsilon$  extraction function defined by Werner in [11]. Essentially, RUSSELL terms do not contain or manipulate logical information attached to objects of subset types, hence we must forget about it when translating.

We define the forgetful map  $()^{\circ}$  as an homomorphism on terms except for the following cases:

$$\begin{aligned} (\{ x : U \mid P \})^{\circ} &= U^{\circ} \\ (\sigma_1 t)^{\circ} &= t^{\circ} \\ (\text{elt } T \ P \ t \ p)^{\circ} &= t^{\circ} \\ (\sigma_2 t)^{\circ} &= \perp \end{aligned}$$

We can now prove the following:

**Theorem 5 (Forgetful map correctness).** *If  $\Gamma \vdash_{CCI} t : T$  then  $\Gamma^\circ \vdash t^\circ : T^\circ$  if  $(\ )^\circ$  is defined (ie, does not return terms containing  $\perp$ ) on  $\Gamma, t$  and  $T$ .*

Practically, this means we can use almost all existing definitions in the COQ environment. This map is involutive and the identity on RUSSELL terms as they do not contain the constructor `elt` or subset projections. Moreover, this erasure function will be defined on all COQ terms elaborated from RUSSELL. Indeed, the second projection of a subset element will only appear in the second component of a subset element. Otherwise, we would have used the projection directly in the RUSSELL term, where it is forbidden. Indeed, in PVS you cannot manipulate proof terms, so you have no way to create a term of type  $0 \neq 0$  from the judgement  $0 : \{ x : \mathbb{N} \mid x \neq 0 \}$ , nor can you actually derive the latter, as it will not type-check. If we allowed the second projection in RUSSELL, that would be doable and we would have an inconsistency. Hence we left only the subset type forming rule (figure 3 on page 6) in RUSSELL, while the introduction and first projection of subset elements are internalized by the coercion judgement.

### 3 From RUSSELL to CIC?

We now build an interpretation  $\llbracket t \rrbracket_\Gamma$  of RUSSELL terms  $t$  in environment  $\Gamma$  into CIC terms. We will check at the end that it respects the  $(\ )^\circ$  operator in the sense that if  $t$  is well-typed in  $\Gamma$  then  $\llbracket t \rrbracket_\Gamma^\circ = t$ .

Our interpretation will build a full-fledged CIC term from its algorithmic skeleton and a rich type written in RUSSELL. Obviously, we cannot infer proof terms where they are needed in CIC, but we can build a *partial* term, leaving typed holes where proofs are needed. Hence we add a rule to build existential variables (or metavariables) in the target calculus ( $\vdash_?$  denotes the new system's typing judgement):

$$\frac{\Gamma \vdash_? P : \mathbf{Prop}}{\Gamma \vdash_? P : P}$$

We restrict it to objects of type `Prop` because we consider that the informative part of the algorithm has been entirely given in the original term.

We are ready to build the interpretation, which should have the following property:

**Proposition 2 (Interpretation correctness).** *If  $\Gamma \vdash_\bullet t : T$  then  $\llbracket \Gamma \rrbracket \vdash_? \llbracket t \rrbracket_\Gamma : \llbracket T \rrbracket_\Gamma$ .*

The proof of this proposition is the main technical contribution of this paper. It also represented the main difficulty in our work. The remainder of this section is organised as follows: first we define the explicit coercion derivation algorithm which will permit to put proof obligations in the terms, then we define the interpretation of terms which is mutually recursive with the previous algorithm. Finally we present a proof sketch of the aforementioned proposition.

### 3.1 Explicit coercions

The derivation of explicit coercions (figure 8) is based on the algorithmic equivalence derivations, hence we can derive a function from the judgement which builds a coercion given two types (denote by  $\mathbf{coerce}_\Gamma T U$ ). The side-conditions of the rules have not changed, so we omit them for better readability. The judgement  $\Gamma \vdash_{CCI} c : T \triangleright_\bullet U : s$  builds a coercion  $c$  from  $T$  to  $U$  in context  $\Gamma$ , that is an object which coerces an object of type  $\llbracket T \rrbracket_\Gamma$  to an object of type  $\llbracket U \rrbracket_\Gamma$  in context  $\llbracket \Gamma \rrbracket$ . We use the algorithmic types to drive the derivation but the resulting object will be a well-typed COQ term (lemma 1).

$$\begin{array}{c}
\triangleright\text{-CONV} \frac{T \equiv_{\beta\pi} U}{\Gamma \vdash_{CCI} \bullet : T \triangleright_\bullet U : s} \quad \triangleright\text{-}\downarrow \frac{\Gamma \vdash_{CCI} c : T^\perp \triangleright_\bullet U^\perp : s}{\Gamma \vdash_{CCI} c : T \triangleright_\bullet U : s} \\
\triangleright\text{-PROD} \frac{\Gamma \vdash_{CCI} c_1 : U \triangleright_\bullet T : s_1 \quad \Gamma, x : U \vdash_{CCI} c_2 : V \triangleright_\bullet W : s_2}{\Gamma \vdash_{CCI} \lambda x : \llbracket U \rrbracket_{\Gamma.c_2}[\bullet (c_1[x])] : \Pi x : T.V \triangleright_\bullet \Pi x : U.W : s_2} \\
\triangleright\text{-SUM} \frac{\Gamma \vdash_{CCI} c_1 : T \triangleright_\bullet U : s \quad \Gamma, x : T \vdash_{CCI} c_2 : V \triangleright_\bullet W : s}{\Gamma \vdash_{CCI} (c_1[\pi_1 \bullet], c_2[\pi_2 \bullet])[\pi_1 \bullet / x]_{\llbracket \Sigma x : U.W \rrbracket_\Gamma} : \Sigma x : T.V \triangleright_\bullet \Sigma x : U.W : s} \\
\triangleright\text{-SUBSET} \frac{\Gamma \vdash_{CCI} c : U \triangleright_\bullet T : \mathbf{Set} \quad \Gamma \vdash_\bullet \{ x : U \mid P \} : \mathbf{Set}}{\Gamma \vdash_{CCI} c[\sigma_1 \bullet] : \{ x : U \mid P \} \triangleright_\bullet T : \mathbf{Set}} \\
\triangleright\text{-PROOF} \frac{\Gamma \vdash_{CCI} c : T \triangleright_\bullet U : \mathbf{Set} \quad \Gamma \vdash_\bullet \{ x : U \mid P \} : \mathbf{Set}}{\Gamma \vdash_{CCI} \text{elt } c \text{ ?}_{\llbracket P \rrbracket_{\Gamma, x : U} [c/x]} : T \triangleright_\bullet \{ x : U \mid P \} : \mathbf{Set}}
\end{array}$$

**Fig. 8.** Coercion derivation

Coercions are formalised as multi-holes evaluation contexts, with  $\bullet$  the denotation of a hole. We define the instantiation  $C[x]$  of a context  $C$  by a term  $x$  as simultaneous substitution of  $x$  for every  $\bullet$  in  $C$ . Hence if  $C$  is an evaluation context and  $x$  a term,  $C[x]$  is a term (i.e., it cannot contain  $\bullet$ ). The holes will denote the object to which the coercion is applied. Other presentations are possible, notably the ‘‘LMS’’ style of [12] which was adopted by Gang Chen in his treatment of coercive subtyping for the Calculus of Constructions [13], and the presentation of Amokrane Saïbi [14]. The LMS style consists in defining a judgement  $x : A \vdash y : B$  which reads:  $A$  is coercible to  $B$  and the coercion is  $\lambda x : A.y$ . This presentation is very elegant and adapted for checking coherence of coercions, which we do not need here as we can get unicity of coercions. Moreover, it did not seem readily usable in our setting, where coercions may contain metavariables. Similarly, Amokrane Saïbi defines a coercion derivation algorithm which respects some coherence criteria, roughly, that type-erased coercions  $\beta\eta$ -reduce to the identity. Again, this was not adapted to our setting, so we invented this new presentation.

As conversion will be preserved by the interpretation  $\llbracket \cdot \rrbracket$ , the rule  $\triangleright\text{-CONV}$  derives an empty coercion, as the target system will be able to derive itself that  $\llbracket T \rrbracket_\Gamma \equiv \llbracket U \rrbracket_\Gamma$ . Similarly for  $\triangleright\text{-}\downarrow$ , we rely on the target system’s conversion rule so that the coercion  $c$  of codomain  $\llbracket U^\perp \rrbracket_\Gamma$  can be seen as an object of type  $\llbracket U \rrbracket_\Gamma$ .

Next, we have the rules for products and sums, which compose coercions. In the  $\succ$ -PROD rule, we first coerce the argument to an object of type  $\llbracket T \rrbracket_{\Gamma}$ . Then we apply it to the coerced function, resulting in an object  $\bullet(c_1[x])$  of type  $\llbracket V \rrbracket_{\Gamma, x: T}[c_1[x]/x]$ . We will see later that this type is equivalent to  $\llbracket V \rrbracket_{\Gamma, x: U}$ , so we can apply the second coercion and get an object of type  $\llbracket W \rrbracket_{\Gamma}$ . There is a little twist when coercing the second component of a dependent sum. Indeed, we must simultaneously substitute the first projection of the coerced sum in the second coercion which was typed in environment  $\Gamma, x : T$  and also instantiate the coercion with the second component. Clearly, the  $\bullet$  of the former shouldn't be instantiated with the later, so we must first instantiate  $c_2$  and then substitute  $\pi_1 \bullet$  for  $x$  ( $x$  cannot appear in  $\pi_2 t$  where  $t : \Sigma x : T. V$ ).

Now for the original part, the  $\succ$ -SUBSET rule simply inserts a projection before applying the inductively defined coercion on the support of the subset. On the other hand,  $\succ$ -PROOF creates an object of a subset using its sole constructor **elt**. The object part is given by the coercion to the support of the target subset while the proof part is promised using a metavariable of type  $\llbracket P \rrbracket_{\Gamma, x: U}[c/x]$  which we will turn later into a proof obligation.

### 3.2 Interpretation of terms

The interpretation of terms from RUSSELL to CIC<sub>?</sub> (figure 9 on the next page) is a straightforward recursive traversal of algorithmic typing derivations. In other, more practical words, we can do interpretation of terms simultaneously with typing in our implementation, just like for the existing coercion system of COQ.

We describe the case of application here; others are similar. First we get the RUSSELL types  $F$  and  $U$  of the function  $f$  and argument  $u$  by calling the typing function on both terms. Then we must ensure that  $F$  can indeed be seen as a product using the  $\mu_{\bullet}()$  operator. We can build the coercion between  $F$  and this product and between  $U$  and its domain. We finally instantiate these coercions by their corresponding interpreted objects and return their application.

### 3.3 A little more expressiveness

One may wonder how can we interpret RUSSELL terms into a system with only  $\beta\pi$ -equivalence and still be conservative with respect to CIC. Indeed if we want the CONV rule to be admissible, we must have for example: If  $\Pi x : A. B \equiv_{\beta\pi} \Pi x : C. D$  then there exists  $c$  such that  $\Gamma \vdash_{CCI} c : \Pi x : A. B \succ_{\bullet} \Pi x : C. D : s$  and  $c \equiv_{\bullet}$ . However, we derive eta-long coercions, so  $c$  will be  $\lambda x : \llbracket C \rrbracket_{\Gamma}. \bullet x$ .

Similarly, we must add  $\eta$  rules for all introduction terms of our language if we want to be conservative and have unicity of coercions. The latter is an important property of our system, as it corresponds in practice to the determinism of proof-obligations generation. The complete equational theory of CIC<sub>?</sub> is given in figure 10 on the facing page. It includes  $\eta$ , surjective pairing for dependent sums and subsets ( $\rho$ ) and proof-irrelevance for the second component of subset objects. Benjamin Werner has studied the addition of proof-irrelevance (rule  $\sigma$ )

$$\begin{aligned}
\llbracket x \rrbracket_\Gamma &= x \\
\llbracket s \rrbracket_\Gamma &= s \quad s \in \{\text{Set}, \text{Prop}, \text{Type}\} \\
\llbracket \Pi x : T.U \rrbracket_\Gamma &= \Pi x : \llbracket T \rrbracket_\Gamma . \llbracket U \rrbracket_{\Gamma, x:T} \\
\llbracket \{ x : U \mid P \} \rrbracket_\Gamma &= \{ x : \llbracket U \rrbracket_\Gamma \mid \llbracket P \rrbracket_{\Gamma, x:U} \} \\
\llbracket \Sigma x : T.U \rrbracket_\Gamma &= \Sigma x : \llbracket T \rrbracket_\Gamma . \llbracket U \rrbracket_{\Gamma, x:T} \\
\llbracket \lambda x : \tau.v \rrbracket_\Gamma &= (\lambda x : \llbracket \tau \rrbracket_\Gamma . \llbracket v \rrbracket_{\Gamma, x:\tau}) \\
\llbracket f \ u \rrbracket_\Gamma &= \text{let } F = \text{type}_\Gamma(f) \text{ and } U = \text{type}_\Gamma(u) \text{ in} \\
&\quad \text{let } (\Pi x : V.W) = \mu_\bullet(F) \text{ in} \\
&\quad \text{let } \pi = \text{coerce}_\Gamma F (\Pi x : V.W) \text{ in} \\
&\quad \text{let } c = \text{coerce}_\Gamma U V \text{ in} \\
&\quad (\pi[\llbracket f \rrbracket_\Gamma]) (c[\llbracket u \rrbracket_\Gamma]) \\
\llbracket (t, u)_{\Sigma x:T.U} \rrbracket_\Gamma &= \text{let } T' = \text{type}_\Gamma(t) \text{ in} \\
&\quad \text{let } ct = \text{coerce}_\Gamma T' T \text{ in} \\
&\quad \text{let } U' = \text{type}_\Gamma(u) \text{ in} \\
&\quad \text{let } cu = \text{coerce}_\Gamma U' U[t/x] \text{ in} \\
&\quad (ct[\llbracket t \rrbracket_\Gamma], cu[\llbracket u \rrbracket_\Gamma])_{\llbracket \Sigma x:T.U \rrbracket_\Gamma} \\
\llbracket \pi_i \ t \rrbracket_\Gamma &= \text{let } T = \text{type}_\Gamma(t) \text{ in} \quad i \in \{1, 2\} \\
&\quad \text{let } \Sigma x : V.W = \mu_\bullet(T) \text{ in} \\
&\quad \text{let } c = \text{coerce}_\Gamma T (\Sigma x : V.W) \text{ in} \\
&\quad \pi_i \ c[\llbracket t \rrbracket_\Gamma]
\end{aligned}$$

**Fig. 9.** Interpretation of terms

in the Calculus of Constructions [11]. It is not a trivial extension and it has far-reaching consequences on the model of the calculus [15]. We direct the reader to these papers for further information on the subject.

$$\begin{aligned}
(\beta) \ (\lambda x : X.e) \ v &\equiv e[v/x] \\
(\pi_i) \ \pi_i \ (e_1, e_2)_T &\equiv e_i \\
(\sigma_i) \ \sigma_i \ (\text{elt } E \ P \ e_1 \ e_2) &\equiv e_i \\
(\eta) \ (\lambda x : X.e \ x) &\equiv e \quad \text{if } x \notin FV(e) \\
(\rho) \ (\pi_1 \ e, \pi_2 \ e)_{\Sigma x:X.Y} &\equiv e \quad \text{if } e : \Sigma x : X.Y \\
&\quad \text{elt } E \ P \ (\sigma_1 \ e) \ (\sigma_2 \ e) \equiv e \quad \text{if } e : \{ x : E \mid P \} \\
(\sigma) \ \text{elt } E \ P \ t \ p &\equiv \text{elt } E \ P \ t' \ p' \ \text{if } t \equiv t'
\end{aligned}$$

**Fig. 10.** Equational theory of  $\text{CIC}_?$

### 3.4 Properties

The correctness proof of the translation is very involved, so we will only sketch it here. A report [16] is available (in French) and a mechanically checked proof is under development.

We first prove reflexivity, symmetry and transitivity of the coercion derivation algorithm, which extends previous proofs for the algorithmic system with

properties on the generated coercions. Then we show substitutivity of the interpretation: if  $\Gamma, x : U, \Delta \vdash t : T$  then  $\llbracket t[u/x] \rrbracket_{\Gamma, \Delta[u/x]} = \llbracket t \rrbracket_{\Gamma, x:U, \Delta} \llbracket [u]_{\Gamma} / x \rrbracket$ . We extend the notion of coercion to contexts next and show stability of judgements under context coercion. It is then possible to prove a lemma about commutativity of substitutions with the interpretation and coercions. Then we get an important corollary: if  $\Gamma \vdash u : U, \Gamma \vdash_{CCI} c : U \triangleright_{\bullet} V : s$  and  $\Gamma, x : V \vdash T : s$  then  $\llbracket T[u/x] \rrbracket_{\Gamma} \equiv \llbracket T \rrbracket_{\Gamma, x:V} [c \llbracket [u]_{\Gamma} \rrbracket / x]$ . Essentially, it shows that coercions inserted by the interpretation depend only on the context and hence could be added later using substitution instead. This allows us to show that equivalence is conserved by interpretation and the following lemma:

**Lemma 1 (Coercion derivation correctness).** *If  $\Gamma \vdash_{CCI} c : T \triangleright_{\bullet} U : s$  then  $\llbracket \Gamma \rrbracket \vdash_{CCI} \lambda x : \llbracket T \rrbracket_{\Gamma}. c[x] : \llbracket T \rrbracket_{\Gamma} \rightarrow \llbracket U \rrbracket_{\Gamma}$ .*

Finally we can show that our interpretation is correct (proposition 2).

## 4 Implementation

This mechanism has been implemented in COQ and tested on simple examples. We actually generate coercions simultaneously with typing in the implementation, which is sound thanks to the previous proofs. Our typing is a clone of COQ's original typing algorithm, so we benefit from all the features of COQ including implicit variables, notations and the existing coercion system. The prototype also contains support for structural and well-founded `Fixpoint` definitions. The type-checker transforms pattern-matching so that an equality between the matched term and the pattern is present in the typing context of each branch and it allows the user to put explicit holes in the term, for example:

```
Program Definition hd ( l : list A | l <> [] ) : A :=
  match l with
  | hd :: tl -> hd
  | [] -> _
  end.
```

Here the second branch can't be taken because we would have  $l = []$  and  $l \neq []$ , but we may not have any placeholder element of type  $A$ .

It must be stressed that we do not lose any information when generating obligations as we have the whole term context at hand, avoiding the complicated proof obligation generation of PVS [3]. Here we would have  $l : \{ l : \text{list } A \mid l \neq [] \}, Heql : \sigma_1 l = []$ . We have tactics that clean the goal and context (deconstructing subsets and simplifying to present  $l : \text{list } A, H : l \neq [], Heql : l = []$  to the user) but it does not match the usability of PVS yet.

## 5 Conclusion

We have developed a new language for writing programs in the COQ proof assistant which allows the user to specify complex programs while keeping the

corresponding code simple. It is a first step towards interpretation of ML code into COQ to build certified programs. While the system and its accompanying proofs may seem complex, the implementation is short, simple and has been tested with success on simple examples. This work is also a second proof of the meaningfulness of the Predicate Subtyping feature of PVS and may help build a common interface for the two provers.

### 5.1 Related work

Other efforts to build certified programs using type theory include CAYENNE [17] and EPIGRAM [18]. CAYENNE offers dependent types and general recursion at the expense of a non-terminating type checker; it can be thought of as a testbed for developing languages with dependent types but is not so much aimed at building certified programs. EPIGRAM ought to become a complete programming language with dependent types at its heart. Instead of using a phase distinction to separate coding and proving, EPIGRAM is based on an interactive, 2-dimensional editing process where code and proofs are written incrementally to get a complete program. Using type annotations and with the help of the editor for structuring, one is able to write programs with precise specifications. However it is not obvious if it could be made usable by programmers because of the paradigm change and scaling issues. The DML language [19] is more akin to comparison with our solution. It provides a way to use a restricted set of dependent types in ML programs, carefully chosen so as to keep type-checking decidable with the help of an automatic prover. Our method should subsume this one as it is perfectly possible to integrate automatic proof tools with COQ to discharge the generated obligations.

### 5.2 Further work

Much has to be done to smoothly integrate RUSSELL into COQ. The treatment of existential variables of COQ has to be improved, the type inference algorithm needs some tuning to become more similar to ML and an integration of proof-irrelevance in the kernel, while rarely needed in practice, is necessary to have a robust system. We also hope to extend this mechanism of proof-obligations generation to other constructs, notably inductive types. Finally, we intend to use this system as a basis for interpretation of complete ML programs in COQ, using a monadic translation or some kind of effects to reflect imperative constructs.

*Acknowledgements* I would like to thank Christine Paulin-Mohring for directing this work and Jean-Christophe Filliâtre for discussions on previous versions of this paper.

## References

1. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development. Springer-Verlag (2004)

2. Shankar, N., Owre, S.: Principles and pragmatics of subtyping in PVS. In Bert, D., Choppy, C., Mosses, P., eds.: *Recent Trends in Algebraic Development Techniques, WADT '99*. Volume 1827 of *Lecture Notes in Computer Science.*, Toulouse, France, Springer-Verlag (1999) 37–52
3. Owre, S., Shankar, N.: The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA (1997)
4. Parent, C.: Synthesizing proofs from programs in the Calculus of Inductive Constructions. In Möller, B., ed.: *MPC*. Volume 947 of *Lecture Notes in Computer Science.*, Springer (1995) 351–379
5. Nordström, B., Petersson, K., Smith, J.M.: *Programming in Martin-Löf's Type Theory*. Oxford University Press (1990)
6. Coquand, T., Huet, G.: The Calculus of Constructions. *Inf. Comp.* **76** (1988) 95–120
7. Chen, G.: *Sous-typage, Conversion de Types et Élimination de la Transitivité*. PhD thesis, Université Paris VII, Laboratoire d'Informatique de l'École Normale Supérieure, Paris (1998)
8. Luo, Z.: Coercive subtyping in type theory. In van Dalen, D., Bezem, M., eds.: *CSL*. Volume 1258 of *Lecture Notes in Computer Science.*, Springer (1996) 276–296
9. Adams, R.: Pure Type Systems with Judgemental Equality. *Journal of Functional Programming* **16** (2006) 219–246
10. Geuvers, H.: A short and flexible proof of strong normalization for the Calculus of Constructions. In Dybjer, P., Nordström, B., Smith, J., eds.: *Selected Papers 2nd Intl. Workshop on Types for Proofs and Programs, TYPES'94*, Båstad, Sweden, 6–10 June 1994. Volume 996 of *LNCS*. Springer-Verlag, Berlin (1995) 14–38
11. Werner, B.: On the strenght of proof-irrelevant type theories. 3rd International Joint Conference on Automated Reasoning (2006)
12. Longo, Milsted, Soloviev: A logic of subtyping. In: *LICS: IEEE Symposium on Logic in Computer Science*. (1995)
13. Chen, G.: Coercive subtyping for the Calculus of Constructions. In: *POPL*. (2003) 150–159
14. Saïbi, A.: Typing algorithm in type theory with inheritance. In: 24th Annual Symposium on Principles of Programming Languages, La Sorbonne, Paris, France, ACM (1997) 292–301
15. Miquel, A., Werner, B.: The not so simple proof-irrelevant model of CC. In Geuvers, H., Wiedijk, F., eds.: *TYPES*. Volume 2646 of *Lecture Notes in Computer Science.*, Springer (2002) 240–258
16. Sozeau, M.: *Coercion par prédicats en Coq*. Master's thesis, Université Paris VII, LRI, Orsay (2005) extended version - <http://www.lri.fr/~sozeau/research/russell/report.pdf>.
17. Augustsson, L.: Cayenne—A language with dependent types. In: *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Baltimore, Maryland. (1998) 239–250
18. McBride, C., McKinna, J.: The view from the left. *J. Funct. Program.* **14**(1) (2004) 69–111
19. Xi, H.: *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania (1998)