Combining the Coq Proof Assistant with First-Order Decision Procedures

Nicolas Ayache[1] and Jean-Christophe Filliâtre[2]

[1] CEA/Saclay
91191 Gif-sur-Yvette Cedex, France
`nicolas.ayache@cea.fr`
[2] CNRS – Université Paris-Sud
Laboratoire de Recherche en Informatique
F-91405 Orsay Cedex, France
`filliatr@lri.fr`

Abstract. We present an integration of first-order automatic theorem provers into the Coq proof assistant. This integration is based on a translation from the higher-order logic of Coq, the Calculus of Inductive Constructions, to a polymorphic first-order logic. This translation is defined and proved sound in this paper. It includes not only the translation of terms and predicates belonging to the first-order fragment, but also several techniques to go well beyond: abstractions of higher-order subterms, case-analysis, mutually recursive functions and inductive types. This process has been implemented in the Coq proof assistant to call the decision procedures Simplify, CVC Lite, haRVey and Zenon through Coq tactics. The first experiments are promising.

1 Introduction

Theorem provers based on highly expressive logics usually lack a good support of proof automation. This is particularly true for the Coq proof assistant [2] which helps the user through very little automation during the proof search. There are two main reasons for this situation. The first reason is that the decision procedures developed for first-order logics does not scale easily to richer logics. The logic behind the Coq proof assistant, for instance, known as the Calculus of Inductive Constructions, intertwines features such as polymorphism, higher-order or dependent types and any of them would require specific adaptation of the first-order techniques. The second reason is that the Coq system is implemented on top of the de Bruijn principle: whatever the way it is built, a proof must eventually be checked by a small and trusted part of the system. This, again, is incompatible with the state of the art decision procedures that usually do not give any kind of justification: most of the time, they simply return a boolean answer.

In this article, we adopt a very pragmatic and modest approach to the challenge of improving proof automation in Coq: we simply accept the above two limitations. Indeed, we are going to interface Coq with external decision procedures that only know about first-order logic and we will trust their results i.e. we give up the de Bruijn principle. Though it may seem very disappointing, there is still a non-trivial task to accomplish, that is the translation from the rich logic of Coq to first-order logic. On one hand, we want this translation to be as powerful as possible and, on the other hand, we do not want the decision procedures to be fooled by awful encodings. This translation process is the main subject of this paper.

Several integrations of higher-order proof assistants and first-order automatic theorem provers have been studied and implemented so far. Most of them focus on resolution-based provers, such as Gandalf integrated into HOL [16, 17] and Martin Löf's Type Theory [5], or the provers Vampire and SPASS integrated into Isabelle/HOL [18]. Such resolution provers take clausal formulae as input, which requires skolemization and conjunctive normal forms to be embedded in the translation process. They may also take some time to decide the given formula when the context is made of hundreds of definitions and lemmas. If it is not an issue for one-shot calls of the provers, as in the TPTP benchmarks [24] and the related competitions, it may become a nuisance when combined with interactive proof. That is why the Isabelle integration [18] opted for provers running in background without (too much) interference with the user's interactive proof.

We are rather interested in decision procedures that can answer within a few seconds and thus can be used during the interactive proof as an alternative to the Coq `auto` tactic. From this point of view, our work is closer to the integration of the ICS decision procedure in PVS [14] (though ICS only handles a quantifier-free fragment of first-order logic) or Harrison's use of tableau techniques in HOL Light [15]. Regarding theories, we only consider equality and linear arithmetic to be relevant, as they are very common in many proofs. The provers we are currently using are Simplify [4], CVC Lite [1], haRVey [21] and Zenon [12] (even if the latter does not handle arithmetic).

These tools do not implement exactly the same logic: Simplify and Zenon implement an *unsorted* logic, haRVey a traditional *many-sorted* logic and CVC Lite a slightly richer typed logic with PVS-like subtypes and even higher-order features. In order to factor out our translation, we need to find a common target logic. Obviously, it needs to be typed. We adopt a *polymorphic first-order logic*. Polymorphism is mandatory to handle large fragments of the Coq context. Indeed, many formalizations, such as the theory of lists in the Coq standard library for instance, are usually done in the most general way and thus polymorphic. Not being able to transmit the corresponding definitions and lemmas to the decision procedures would be unfortunate. Then going from polymorphic first-order logic to the input logics of the various provers is considered as a separate step in our integration, for which we currently delegate to the Why tool [13].

Our work is not the first integration of first-order provers into the Coq proof assistant. We can mention an interface to the resolution prover Bliksem [8], the theorem prover for first-order intuitionistic logic JProver [23] and the Elan rewriting system [6]. Less related to our work is a direct implementation of first-order proof search in Coq by Corbineau [10].

This paper is organized as follows. First, Section 2 briefly introduces the source and target logics, namely the Calculus of Inductive Constructions and a polymorphic first-order logic. Then Section 3 defines our translation from the former to the latter and proves its soundness. Finally, Section 4 details the implementation in the Coq proof assistant and reports on the first experiments.

2 Logics

2.1 The Calculus of Inductive Constructions

The logic behind the Coq proof assistant is the Calculus of Inductive Constructions [9, 19], written CIC in the following. It is a typed λ-calculus with polymorphism, higher-order, dependent types and a primitive notion of inductive types. The Coq proof assistant relies on the Curry-Howard isomorphism: a proposition is a type of the CIC and a proof of this proposition is a λ-term inhabitating this type.

The CIC has a single syntactic category used for both types and terms and defined as follows:

$$\begin{array}{rcl} s & ::= & \texttt{Set} \mid \texttt{Prop} \mid \texttt{Type}_i \\ t & ::= & s \mid x \mid c \mid C \mid I \\ & \mid & \forall x : t,\, t \mid \lambda x : t,\, t \mid t\ t \mid \texttt{case}(t, t, t, \ldots, t) \\ & \mid & \texttt{fix}\ x\ \{x : t := t; \ldots; x : t := t\} \end{array}$$

where x ranges over variable names, c over constant names, C over constructor names and I over inductive type names. `Set` and `Prop` should be thought of as the sorts for datatypes and propositions respectively, and $\texttt{Type}_i$ as the sorts of everything above in the type hierarchy. The dependent product $\forall x : t_1,\, t_2$ is written $t_1 \rightarrow t_2$ whenever x does not appear in t_2 and then can be seen as the function type or as the logical implication, depending on which side of the Curry-Howard isomorphism one is considering. The term $\texttt{case}(e, P, t_1, \ldots, t_n)$ is a powerful elimination construct that reduces to the branch t_i whenever e reduces to the i-th constructor of some inductive type (P is a term giving the type of the result). The term $\texttt{fix}\ x_k\ \{(x_i : T_i := t_i)_{i=1,\ldots,n}\}$ stands for the k-th function of a block of n mutually recursive functions.

A CIC context Γ is a list of declarations that can be of three kinds:

- a variable declaration $x : t$,
- a constant definition $c := t : t$,
- a set of mutually inductive types definitions

$$\texttt{Ind}(I : t := C : t | \ldots | C : t;\ \ \ldots;\ I : t := C : t | \ldots | C : t)$$

Example 1. Here is a CIC context introducing Peano's natural numbers as an inductive datatype *nat* and an addition function *plus* recursively defined on its first argument:

$$\begin{array}{l}
\texttt{Ind}(nat : \texttt{Set} := O : nat \mid S : nat \to nat) \\
plus \ := \ \texttt{fix}\ f\ \{f : nat \to nat \to nat := \lambda x : nat,\ \lambda y : nat, \\
\qquad\qquad\qquad \texttt{case}(x, (\lambda\_ : nat,\ nat), y, \lambda z : nat,\ S\ (f\ z\ y))\} \\
\qquad : nat \to nat \to nat
\end{array}$$

Giving the CIC typing rules would go far beyond the scope of this paper. The rules corresponding to what is implemented in the Coq proof assistant can be found in the chapter 4 of the Coq reference manual [2] or in the Coq'Art [7]. The complexity of the CIC typing is mainly due to a convertibility rule that allows arbitrary reductions (i.e. computations) to be performed in types. Such reductions necessarily terminate since the CIC enjoys a normalization property. In the following, we write $\mathsf{nf}(t)$ for the normal form of a term t.

2.2 Polymorphic First-Order Logic

Our target language is a Polymorphic First-Order Logic, written PFOL in the following. As usual with first-order settings, we introduce successively the syntactic notions of *types*, *terms* and *predicates*.

A *type* is introduced by its name s (an identifier) and its arity n (a non-negative integer) that is its number of type parameters. When $n > 0$ the type is polymorphic (i.e. it is a type operator) and when $n = 0$ the type is monomorphic. Then *type expressions* τ are defined as follows, where s ranges over the set of type names and α over an infinite set of type variables:

$$\textit{types} \qquad \tau \ ::= \ \alpha \mid s[\tau, \dots, \tau]$$

When a type s is monomorphic, we write directly s instead of $s[]$. As usual, terms are built from variables and function symbols f and predicates are built from predicate symbols p and the usual first-order connectives:

$$\begin{array}{lrcl}
\textit{terms} & t & ::= & x \mid f(t, \dots, t) \\
\textit{predicates} & P & ::= & \top \mid \bot \mid P \wedge P \mid P \vee P \mid \neg P \mid P \Rightarrow P \\
 & & \mid & p(t, \dots, t) \mid \forall x : \tau.\, P \mid \exists x : \tau.\, P
\end{array}$$

Constants are functions with an arity 0. Finally, a *theory* Σ is a list of declarations δ of types, variables, functions, predicates and axioms:

$$\begin{array}{rcl}
\delta & ::= & \texttt{type}\ s[n] \mid x : \tau \mid \texttt{fun}\ f : \forall \boldsymbol{\alpha}.\, \tau, \dots, \tau \to \tau \\
 & \mid & \texttt{pred}\ p : \forall \boldsymbol{\alpha}.\, \tau, \dots, \tau \mid \texttt{axiom}\ \forall \boldsymbol{\alpha}.\, P
\end{array}$$

The notation $\forall \boldsymbol{\alpha}$ stands for the quantification over a (possibly empty) set of type variables. This is precisely where the polymorphism is introduced: functions, predicates and axioms may all be polymorphic. These syntactic categories being set, we can now introduce the following typing judgments:

$\Sigma \vdash \tau$ wf	the type τ is well-formed in Σ
$\Sigma \vdash t : \tau$	the term t is well-typed in Σ, of type τ
$\Sigma \vdash P$ wf	the predicate P is well-formed in Σ
$\vdash \Sigma$ wf	the theory Σ is well-formed

The rules defining these judgments are rather straightforward and gathered in Appendix A.

We assume some predefined notions of equality and linear integer arithmetic, that is: a polymorphic equality with the usual infix notation $t_1 = t_2$; a monomorphic type *int* for the integers; the infinite set of integer constants $\ldots,-2,-1,0,1,2,\ldots$ of type *int*; addition and subtraction function symbols with the usual infix notations $+$ and $-$; and inequality predicates with the usual infix notations $<, \leq, >$ and $\geq$.

Finally, validity is introduced as the judgement $\Sigma \models P$ meaning "P is provable in the theory Σ". It is defined by a set of natural deduction rules given in Appendix A.

3 From CIC to PFOL

In order to call a first-order decision procedure on the current goal from the Coq toplevel, we need to translate both the context and the goal from CIC to PFOL. This section defines such a translation and proves its soundness.

3.1 The translation

The translation is defined as a bunch of functions translating respectively types, terms, predicates and contexts. All these functions are given here in pseudo-code in Figures 1–5. They are partial functions i.e. they may fail (when the CIC term has no PFOL counterpart). Failures are handled implicitly: when a function is not defined, it is assumed to fail, and whenever a function call fails, we implicitly jump to the next case of the function being defined. We now detail the various translation functions.

Translating types. The function tr-type(Σ, v, t) translates a CIC term t of type `Set` into a PFOL type expression τ. Σ is a PFOL theory (it must be seen as the translation of the CIC environment so far) and v is the set of type variables α of type `Set` that may appear in t and thus in τ. The definition of tr-type is given Figure 1. Notice that we compute the normal form of t before performing the translation (using the function nf).

Translating terms. The function tr-term(Σ, t) translates a CIC term t of a type T itself of type `Set` into a PFOL term. Σ is a PFOL theory. The definition of tr-term is given Figure 2. The function abstract(t) used in tr-term is replacing a CIC term t by a new variable, provided that t is closed.

```
{ $\textit{assumption } t : \texttt{Set}$ }
tr-type($\Sigma$,$v$,$t$) =
  let $t$ = nf($t$) in
  if $t$ is a variable $\alpha$ in $v$ then return $\alpha$
  if $t = s\ t_1\ \dots\ t_n$ and type $s[n] \in \Sigma$ then
    return $s$[tr-type($\Sigma$,$v$,$t_1$),...,tr-type($\Sigma$,$v$,$t_n$)]
```

Fig. 1. Translating types

```
{ $\textit{assumption } t : T : \texttt{Set}$ }
tr-term($\Sigma$,$t$) =
  if $t$ is a variable or a constant $x$ bound in $\Sigma$ then return $x$
  if $t$ is an integer constant $n$ then return $n$
  if $t = \texttt{plus}\ t_1\ t_2$ then return tr-term($\Sigma$,$t_1$)+tr-term($\Sigma$,$t_2$)
  if $t = \texttt{minus}\ t_1\ t_2$ then return tr-term($\Sigma$,$t_1$)−tr-term($\Sigma$,$t_2$)
  if $t = f\ c_1\ \dots\ c_n$ where $f$ is a global and $n \geq 1$ then
    if fun $f$:$\forall\alpha_1\dots\alpha_k.\tau_1,\dots,\tau_{n-k} \to \tau \in \Sigma$ then
      return $f$(tr-term($\Sigma$,$c_{k+1}$),...,tr-term($\Sigma$,$c_n$))
    else
      let $f_0$ = abstract($f\ c_1$) in return tr-term($\Sigma$,$f_0\ c_2\ \dots\ c_n$)
```

Fig. 2. Translating terms

Translating predicates. The function tr-pred(Σ, v, t) translates a CIC term t of type `Prop` into a PFOL predicate. Σ is a PFOL theory and v is the set of type variables that may be bound in t. In the CIC, all the logical connectives (apart from universal quantification) are not primitive but defined using inductive types. In order to translate them to the corresponding PFOL connectives, we recognize these constants (namely `True`, `False`, `not`, `and`, `or` and `ex`). The definition of tr-pred is given Figure 3.

Translating environments. The translation of environments is based on a main function tr-decl(Σ, x, t) that translates the CIC declaration of x of type t into a PFOL declaration, that is either a type, a function, a predicate or an axiom declaration. Note that x may be either a CIC variable, a constant, an inductive type or a constructor. The function tr-decl handles the polymorphism by extracting the prenex type quantifications $A_1, \dots, A_k$ and the monomorphic declarations as a particular case when $k = 0$. Similarly, it handles the case of constants whenever $n = 0$.

Finally, the translation of a CIC environment Γ in a PFOL theory Σ is realized by the function tr-env which is processing all the declarations in Γ one by one using tr-decl. Of course, some of them may not be first-order and thus will not be considered. The definition of tr-env is given Figure 5.

The PFOL theory Σ resulting from the translation of a CIC context Γ contains enough information for the translation of a CIC proposition to type-check, as we will show in Section 3.3. We can however transfer more information from

```
{ assumption $t$ : Prop }
tr-pred($\Sigma$,$v$,$t$) =
  if $t = $ eq $T$ $t_1$ $t_2$ then return tr-term($\Sigma$,$t_1$)=tr-term($\Sigma$,$t_2$)
  if $t = $ lt $t_1$ $t_2$ then return tr-term($\Sigma$,$t_1$)<tr-term($\Sigma$,$t_2$)
  if $t = $ ... (similar for other arithmetic comparisons) ...
  if $t = $ True then return $\top$
  if $t = $ False then return $\bot$
  if $t = $ not $q$ then return $\neg$ tr-pred($\Sigma$,$v$,$q$)
  if $t = $ and $p$ $r$ then return tr-pred($\Sigma$,$v$,$q$) $\wedge$ tr-pred($\Sigma$,$v$,$r$)
  if $t = $ or $p$ $r$ then return tr-pred($\Sigma$,$v$,$q$) $\vee$ tr-pred($\Sigma$,$v$,$r$)
  if $t = q \rightarrow r$ then return tr-pred($\Sigma$,$v$,$q$) $\Rightarrow$ tr-pred($\Sigma$,$v$,$r$)
  if $t = $ ex $T$ $(\lambda x : T,\ q)$ then
    let $\tau = $ tr-type($\Sigma$,$v$,$T$) in return $\exists x : \tau.$ tr-pred($\Sigma + \{x : \tau\}$,$v$,$q$)
  if $t = \forall x : T,\ q$ then
    let $\tau = $ tr-type($\Sigma$,$v$,$T$) in return $\forall x : \tau.$ tr-pred($\Sigma + \{x : \tau\}$,$v$,$q$)
  if $t = p\ c_1\ \dots\ c_n$ where $p$ is a global
  and pred $p$:$\forall\alpha_1 \dots \alpha_k.\tau_1, \dots, \tau_{n-k} \in \Sigma$ then
    return $p$(tr-term($\Sigma$,$c_{k+1}$),...,tr-term($\Sigma$,$c_n$))
```

Fig. 3. Translating predicates

```
{ assumption $x : t$ }
tr-decl($\Sigma$,$x$,$t$) =
  let $A_1, \dots, A_k, T$ such that $t = \forall A_1$ : Set, $\dots, \forall A_k$ : Set, $T$ in
  if $T = $ Set then return type $x[k]$
  let $v = \{A_1; \dots; A_k\}$ in
  if $T = T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_{n+1}$ with the $T_i$ of type Set then
    let $\tau_1 = $ tr-type($\Sigma$,$v$,$T_1$) and ... and $\tau_{n+1} = $ tr-type($\Sigma$,$v$,$T_{n+1}$) in
    return fun $x : \forall \boldsymbol{A}.\,\tau_1, \dots, \tau_n \rightarrow \tau_{n+1}$
  if $T = T_1 \rightarrow \cdots \rightarrow T_n \rightarrow$ Prop with the $T_i$ of type Set then
    let $\tau_1 = $ tr-type($\Sigma$,$v$,$T_1$) and ... and $\tau_n = $ tr-type($\Sigma$,$v$,$T_n$) in
    return pred $x : \forall \boldsymbol{A}.\,\tau_1, \dots, \tau_n$
  if $T$ is of type Prop then return axiom $\forall \boldsymbol{A}.$ tr-pred($\Sigma$,$v$,$T$)
```

Fig. 4. Translating declarations

```
tr-env($\Gamma$) =
  $\Sigma := \emptyset$
  for each declaration $\delta$ in $\Gamma$ do
    if $\delta$ is $x : t$ then $\Sigma := \Sigma +$ tr-decl($\Sigma$,$x$,$t$)
    if $\delta$ is $c := t : T$ then $\Sigma := \Sigma +$ tr-decl($\Sigma$,$x$,$T$)
    if $\delta$ is $\mathtt{Ind}\big(I_i : T_i := (C_{i,j} := t_{i,j})_{j=1,\dots,k_i}\big)_{i=1,\dots,n}$ then
      for $i = 1, \dots, n$ do $\Sigma := \Sigma +$ tr-decl($\Sigma$,$I_i$,$T_i$)
      for $i = 1, \dots, n$ do
        for $j = 1, \dots, k_i$ do $\Sigma := \Sigma +$ tr-decl($\Sigma$,$C_{i,j}$,$t_{i,j}$)
  return $\Sigma$
```

Fig. 5. Translating environments

CIC to PFOL by also translating constant definitions and some properties of inductive types whenever possible.

Translating definitions. If a CIC definition $c := t : T$ is translated into a PFOL function $\texttt{fun}\ f : \forall \boldsymbol{\alpha}.\, \tau_1, \ldots, \tau_n \to \tau$ or a predicate $\texttt{pred}\ p : \forall \boldsymbol{\alpha}.\, \tau_1, \ldots, \tau_n$, we also try to interpret the definition body t as much as possible. We distinguish the two cases of a non-recursive and of a recursive definition:

– non-recursive definition: performing some η-expansions if necessary, we can always put the body t into the form

$$t = \lambda \alpha_1 : \texttt{Set} \ldots \lambda \alpha_k : \texttt{Set},\ \lambda x_1 : T_1 \ldots \lambda x_n : T_n,\ b$$

with $\textsf{tr-type}(\Sigma, \boldsymbol{\alpha}, T_i) = \tau_i$. Then we (try to) append the following axiom to Σ in the case of a function f:

$$\forall \boldsymbol{\alpha}.\, \forall \boldsymbol{x}.\, f(x_1, \ldots, x_n) = \textsf{tr-term}(\Sigma + \{x_1 : \tau_1, \ldots, x_n : \tau_n\}, b)$$

and the following axiom in the case of a predicate p:

$$\forall \boldsymbol{\alpha}.\, \forall \boldsymbol{x}.\, p(x_1, \ldots, x_n) \Leftrightarrow \textsf{tr-pred}(\Sigma + \{x_1 : \tau_1, \ldots, x_n : \tau_n\}, \boldsymbol{\alpha}, b)$$

(where $P \Leftrightarrow Q$ is simply a shortcut for $P \Rightarrow Q \wedge Q \Rightarrow P$).

– recursive definition: we only consider the case of a recursive definition of the shape

$$\begin{aligned} f := \texttt{fix}\ f\ \{\ f : T := \ & \lambda \alpha_1 : \texttt{Set}, \ldots \lambda \alpha_k : \texttt{Set}, \\ & \lambda x_1 : T_1, \ldots \lambda x_n : T_n, \\ & \texttt{case}(x_i, P, (\lambda \boldsymbol{y_1},\ t_1), \ldots, (\lambda \boldsymbol{y_m},\ t_m))\ \} \end{aligned}$$

i.e. a recursive function performing an immediate case-analysis over one of its arguments. Fortunately, this is the most common situation. Then we (try to) append one axiom for each branch t_i, that is

$$\begin{aligned} & \forall \boldsymbol{\alpha}.\, \forall x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.\, \forall \boldsymbol{y_i}. \\ & \quad f(x_1, \ldots, x_{i-1}, C_i(\boldsymbol{y_i}), x_{i+1}, \ldots, x_n) = \textsf{tr-term}(\Sigma + \boldsymbol{x} + \boldsymbol{y_i}, t_i) \end{aligned}$$

Translating Inductive Types Properties. For each individual inductive type $I : t := C_1 : t_1 \mid \ldots \mid C_n : t_n$ such that $\textsf{tr-decl}(\Sigma, I, t) = \texttt{type}\ s[n]$ we append to the PFOL theory the following axioms expressing that I is the free algebra generated by the constructors $C_1, \ldots, C_n$:

– (inversion)

$$\forall \boldsymbol{\alpha}.\, \forall x : I.\, (\exists \boldsymbol{y_1}.\, x = C_i(\boldsymbol{y_1})) \vee \cdots \vee (\exists \boldsymbol{y_n}.\, x = C_n(\boldsymbol{y_n}))$$

– (injection) For each non-constant constructor C_i,

$$\forall \boldsymbol{\alpha}.\, \forall \boldsymbol{y}.\, \forall \boldsymbol{y'}.\, C_i(\boldsymbol{y}) = C_i(\boldsymbol{y'}) \Rightarrow y_1 = y'_1 \wedge \cdots \wedge y_{k_i} = y'_{k_i}$$

– (free algebra) For each pair of constructors C_i, C_j with $i \neq j$,

$$\forall \boldsymbol{\alpha}.\, \forall \boldsymbol{y}.\, \forall \boldsymbol{y'}.\, C_i(\boldsymbol{y}) \neq C_j(\boldsymbol{y'})$$

We cannot express that this is the *smallest* free algebra, however, since this is not expressible in first-order logic.

3.2 Example

The CIC context defining *nat* and *plus* in Example 1 page 4 is translated into the following theory:

$$\begin{array}{rcl}
\Sigma & := & \texttt{type}\ nat[0] \\
 & & \quad \texttt{fun}\ O :\to nat \\
 & & \quad \texttt{fun}\ S : nat \to nat \\
 & & \texttt{axiom}\ \forall x : nat.\, x = O \vee \exists y : nat.\, x = S(y) \\
 & & \texttt{axiom}\ \forall x, y : nat.\, S(x) = S(y) \Rightarrow x = y \\
 & & \texttt{axiom}\ \forall x : nat.\, O \neq S(x) \\
 & & \quad \texttt{fun}\ plus : nat \to nat \to nat \\
 & & \texttt{axiom}\ \forall y : nat.\, plus(O, y) = y \\
 & & \texttt{axiom}\ \forall z, y : nat.\, plus(S(z), y) = S(plus(z, y))
\end{array}$$

3.3 Soundness

This section establishes that our translation is sound. First, it is clear that the translation is terminating, since the size of the main argument in tr-type, tr-term and tr-pred is decreasing on each recursive call. Second, we give type soundness results stating that the various PFOL entities obtained by translation are well-formed. In the following, Γ is a CIC context and $\Sigma = \text{tr-env}(\Gamma)$.

Lemma 1 (type soundness of tr-type). *Let t be a CIC term of type* `Set` *in Γ and $\boldsymbol{\alpha}$ its variables of type* `Set`*. If tr-type$(\Sigma, \boldsymbol{\alpha}, t) = \tau$ then $\Sigma \vdash \tau$* wf*.*

Proof. Straightforward by induction on the size of t.

Lemma 2 (type soundness of tr-term). *Let t be a CIC term of type T itself of type* `Set` *in Γ. If tr-term$(\Sigma, t) = t'$ then there exists τ such that $\Sigma \vdash t' : \tau$ and $\tau =$ tr-type(Σ, v, T) where v is the set of variables of type* `Set` *in T.*

Proof. By induction on the size of t. To apply the induction hypothesis, we need the property that two convertible CIC types are translated to the same PFOL type, which is ensured by the normalization of types in tr-type.

Lemma 3 (type soundness of tr-pred). *Let t be a CIC term of type* `Prop` *in Γ and $\boldsymbol{\alpha}$ be the variables of t of type* `Set`*. If tr-pred$(\Sigma, \boldsymbol{\alpha}, t) = P$ then $\Sigma \vdash P$* wf*.*

Proof. By induction on the size of t. Proof similar to the one of Lemma 2.

Lemma 4 (type soundness of tr-env). *Let Γ be a CIC environment and $\Sigma =$ tr-env(Γ). Then $\vdash \Sigma$* wf*.*

Proof. By induction on Γ and by case analysis on tr-decl. Then it is straightforward using Lemma 1 and Lemma 3.

Finally, we can establish provability soundness: if the resulting PFOL predicate is provable in the resulting PFOL theory then the CIC proposition is provable in the CIC context.

Theorem 1 (soundness). *Let t be a CIC term of type* `Prop` *in Γ, $\boldsymbol{\alpha}$ be the variables of t of type* `Set` *and $\Sigma = \textsf{tr-env}(\Gamma)$. If $\textsf{tr-pred}(\Sigma, \boldsymbol{\alpha}, t) = P$ and $\Sigma \models P$ then t is classically provable in Γ i.e. there exists a CIC term π of type t in $\Gamma + EM : \forall P : \texttt{Prop}.P \vee \neg P$.*

Proof. By induction on the derivation of $\Sigma \models P$. Each case is justified by a CIC proof term:

- for the $\textsf{Ax}$ rule, we must build a proof term for each proposition inserted as an axiom in Σ through our translation. The axioms related to functions (being recursive or not) and predicates definitions are justified by the CIC reduction rules. The properties inherent to the inductive types (namely inversion, injection, and free algebra) can be proved as lemmas using the appropriate Coq tactics [11];
- the $\textsf{EM}$ rule corresponds to an instance of the axiom EM added to Γ;
- every PFOL proposition proved through the $\textsf{Arith}$ rule can be proved in Coq using the `omega` tactic [20];
- the remaining deduction rules are easily simulated in CIC using the definition of the connectives and the induction hypothesis. To apply the induction hypothesis, we need to exhibit a CIC term for each PFOL sub-formula appearing in the premises, which is a consequence of the following three lemmas:

Lemma 5 (surjection of tr-type). *Let τ be a PFOL type. If $\Sigma \vdash \tau$ $\textsf{wf}$ then there exists a CIC term t of type* `Set` *such that $\textsf{tr-type}(\Sigma, \boldsymbol{\alpha}, t) = \tau$ where $\boldsymbol{\alpha}$ are the type variables of t.*

Proof. By induction on the derivation of $\Sigma \vdash \tau$ $\textsf{wf}$. It requires the extra property that for any `type` $s[n] \in \Sigma$ there is a CIC global s of type $\forall A_1, \ldots, A_n : \texttt{Set}, T$ with T of type `Set` in $\Gamma, A_1, \ldots, A_n$ (definition of $\textsf{tr-decl}$).

Lemma 6 (surjection of tr-term). *Let t' be a PFOL term, τ a PFOL type, T a CIC type and $\boldsymbol{\alpha}$ the variables of T of type* `Set`*. If $\Sigma \vdash t' : \tau$ and $\textsf{tr-type}(\Sigma, \boldsymbol{\alpha}, T) = \tau$ then there exists a CIC term t of type T in Γ such that $\textsf{tr-term}(\Sigma, t) = t'$.*

Proof. By induction on the size of t'.

Lemma 7 (surjection of tr-pred). *Let P' be a PFOL predicate. If $\Sigma \vdash P'$ $\textsf{wf}$ then there exists a CIC proposition P such that $\textsf{tr-pred}(\Sigma, \boldsymbol{\alpha}, P) = P'$ where $\boldsymbol{\alpha}$ are the type variables of P'.*

Proof. By induction on the derivation of $\Sigma \vdash P'$ $\textsf{wf}$.

4 Implementation

The work presented in this paper has been implemented in the Coq proof assistant [2] to call the decision procedures Simplify [4], CVC Lite [1], Zenon [12] and haRVey [21] through Coq tactics. Such tactics proceed in two steps, as illustrated

Figure 6. First, the Coq context and the current goal are translated to a context and a goal in PFOL, independently of the decision procedure to be used, in the input syntax of the Why tool [13].

Then the Why tool is used to produce an input file for the particular prover to be used. In particular, this is the Why tool which is responsible for translating PFOL to an unsorted first-order logic (for Simplify and Zenon) or to a "monomorphic" first-order logic (for CVC Lite and haRVey). This is not an obvious step (see for instance the discussions in [5, 15, 18]), but this is not the subject of this paper.

Finally, the selected prover is called (with a timeout, typically 10 seconds) and whenever it validates the goal, an axiom is built in Coq corresponding to the current Coq goal and is used to discharge it.

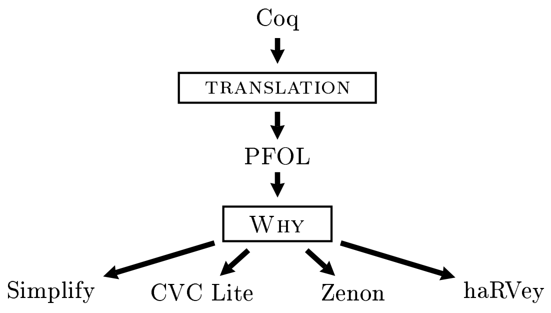

Fig. 6. Calling decision procedures from Coq

The Why tool is available independently of the Coq proof assistant. The remaining of the implementation, namely the translation described in this paper, is implemented in Objective Caml [3] and integrated to the Coq sources. It amounts to less than 600 lines of code, which is fairly small for an integration of four different decision procedures within a proof assistant. Compared to the theoretical presentation of this paper, it adds a lazy strategy that only translates the pieces of the context that are needed, and a cache mechanism that remembers these translations from one call to another.

The first experiments are promising. We took several existing Coq proofs and tried to replace pieces of proof scripts by calls to external provers as much as possible. In most cases, the Coq tactics `auto`, `intuition`, `omega` and `firstorder` could be replaced by the new tactics `zenon` or `simplify`. More significatively, combinations of several tactics including applications of transitivity lemmas and rewritings could be replaced by a single call to a first-order prover. Interestingly, the two provers Zenon and Simplify appear to be complementary. In particular, even if Zenon does not support arithmetic it sometimes succeeds where Simplify fails. In practice, we observed one successful call to an external prover each 3 lines of tactics, with one call to Simplify for two calls to Zenon. The haRVey and

CVC Lite provers were tested too, but without significative improvement with respect to Zenon or Simplify.

This implementation is currently available only as part of the development version of Coq, which is accessible by anonymous CVS access (see `coq.inria.fr` for details).

5 Conclusion

We have presented an integration of four decision procedures into the Coq proof assistant, based on a pragmatic and sound translation from the Calculus of Inductive Constructions to a polymorphic first-order logic. Though the first experiments are very promising, this work could be improved in many ways.

First, we could interface other decision procedures. This is rather easy, since it only requires a support for the new provers by the Why tool. This is even immediate for provers accepting the SMT-LIB input format [22], for that it is already supported by Why. However, the issue is rather to find a prover powerful enough, that is handling the full first-order logic, equality and arithmetic.

Second, we should translate proofs generated by the decision provers back to Coq proof terms, when available. Currently, only Zenon and CVC Lite are able to produce such proofs. Zenon can even generate a script of Coq tactics or a Coq proof term, so the translation back is immediate (this is not yet implemented though). CVC Lite produces proof traces in its own format but due to lack of documentation on this format, we did not manage to translate them into Coq proofs.

The JProver [23] fulfills the two requirements above (it handles intuitionistic first-order logic and produces proofs) and thus is a good candidate for an integration into our framework. Actually, the JProver has already been interfaced with Coq, by Huang Guan-Shieng in 2002, but it would benefit from our translation that covers a larger part of the context than its current integration.

Finally, we could use several other techniques to translate a wider fragment of the Coq logic, such as recognizing the usual logical connectives in user-defined inductive predicates (as already done by other Coq tactics such as `intuition` or `firstorder` [10]) or generating axioms for the inversion of inductive predicates [11].

References

1. CVC Lite Homepage. `http://verify.stanford.edu/CVCL`.
2. The Coq Proof Assistant. `http://coq.inria.fr/`.
3. The Objective Caml language. `http://caml.inria.fr/`.
4. The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.
5. Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a Logical Framework to a First-Order Logic Prover. In Bernhard Gramlich, editor, *5th International Workshop on Frontiers of Combining Systems, FroCoS'05, Vienna, Austria, September 19-21, 2005*, Lecture Notes in Computer Science. Springer Verlag, 2005.

6. Cuihtlauac Alvarado and Quang-Huy Nguyen. Elan for equational reasoning in coq. In *Proceeding of the 2nd Workshop on Logical Frameworks and Metalanguages*, Santa Barbara, California, 2000.
7. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`.
8. Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *J. Autom. Reasoning*, 29(3-4):253–275, 2002.
9. Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.
10. Pierre Corbineau. First-order reasoning in the Calculus of Inductive Constructions. In *TYPES 2003 : Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 162–177. Springer-Verlag, 2004.
11. Cristina Cornes and Delphine Terrasse. Automating Inversion of Inductive Predicates in Coq. In *Types for Proofs and Programs (TYPES'95)*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer-Verlag, 1996.
12. Damien Doligez. The Zenon prover. Distributed with the Focal Project at `http://focal.inria.fr/`.
13. J.-C. Filliâtre. The Why verification tool. `http://why.lri.fr/`.
14. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249, 2001.
15. John Harrison. First Order Logic in Practice. In *International Workshop on First-Order Theorem Proving (FTP'97)*, Linz (Austria), 1997.
16. Joe Hurd. Integrating Gandalf and HOL. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, LNCS. Springer-Verlag, 1999.
17. Joe Hurd. An LCF-Style Interface between HOL and First-Order Logic. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 134–138, London, UK, 2002. Springer-Verlag.
18. Jia Meng, Claire Quigley, and L. C. Paulson. Automation for Interactive Proof: First Prototype. *Information and Computation*, 2005. In press.
19. Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.
20. William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
21. Silvio Ranise and David Déharbe. The haRVey decision procedure. `http://www.loria.fr/~ranise/haRVey/`.
22. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. In *PDPAR'03*, July 2003. `http://goedel.cs.uiowa.edu/smtlib/`.
23. Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating Connection-Based Theorem Proving into Interactive Proof Assistants. *Lecture Notes in Computer Science*, 2083:421+, 2001.
24. Geoff Stutcliffe and Christian Suttner. The TPTP problem library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

A Polymorphic First-Order Logic

Types (τ), terms (t) and predicates (P) are built according to the following grammars

$$\begin{array}{rcl} \tau & ::= & \alpha \mid s[\tau,\ldots,\tau] \\ t & ::= & x \mid f(t,\ldots,t) \\ P & ::= & p(t,\ldots,t) \\ & \mid & \top \mid \bot \mid P \wedge P \mid P \vee P \mid \neg P \mid P \Rightarrow P \\ & \mid & \forall x : \tau.\, P \mid \exists x : \tau.\, P \end{array}$$

A theory Σ is a finite list of declarations δ where

$$\begin{array}{rcl} \delta & ::= & \texttt{type}\ s[n] \mid x : \tau \mid \texttt{fun}\ f : \forall \boldsymbol{\alpha}.\, \tau,\ldots,\tau \to \tau \\ & \mid & \texttt{pred}\ p : \forall \boldsymbol{\alpha}.\, \tau,\ldots,\tau \mid \texttt{axiom}\ \forall \boldsymbol{\alpha}.\, P \end{array}$$

Well-formed types ($\Sigma \vdash \tau\ \mathsf{wf}$):

$$\frac{}{\Sigma \vdash \alpha\ \mathsf{wf}}(\mathsf{Ty}_1) \qquad \frac{\texttt{type}\ s[n] \in \Sigma \quad \forall i,\ \Sigma \vdash \tau_i\ \mathsf{wf}}{\Sigma \vdash s[\tau_1,\ldots,\tau_n]\ \mathsf{wf}}(\mathsf{Ty}_2)$$

Well-typed terms ($\Sigma \vdash t : \tau$):

$$\frac{x : \tau \in \Sigma}{\Sigma \vdash x : \tau}(\mathsf{T}_1) \qquad \frac{\begin{array}{c}\texttt{fun}\ f : \forall \boldsymbol{\alpha}.\, \tau_1,\ldots,\tau_n \to \tau \in \Sigma \\ \mathsf{Subst}(\sigma,\Sigma) \quad \forall i,\ \Sigma \vdash t_i : \sigma(\tau_i)\end{array}}{\Sigma \vdash f(t_1,\ldots,t_n) : \sigma(\tau)}(\mathsf{T}_2)$$

where σ is a mapping from type variables to types, naturally extended to types with $\sigma(s[\tau_1,\ldots,\tau_n]) = s[\sigma(\tau_1),\ldots,\sigma(\tau_n)]$. We write $\mathsf{Subst}(\sigma,\Sigma)$ whenever $\Sigma \vdash \sigma(\alpha)\ \mathsf{wf}$ holds for any type variable α.

Well-typed predicates ($\Sigma \vdash P\ \mathsf{wf}$):

$$\frac{\texttt{pred}\ p : \forall \boldsymbol{\alpha}.\, \tau_1,\ldots,\tau_n \in \Sigma \quad \mathsf{Subst}(\sigma,\Sigma) \quad \forall i,\ \Sigma \vdash t_i : \sigma(\tau_i)}{\Sigma \vdash p(t_1,\ldots,t_n)\ \mathsf{wf}}(\mathsf{P}_1)$$

$$\frac{}{\Sigma \vdash \top\ \mathsf{wf}}(\mathsf{P}_2) \qquad \frac{}{\Sigma \vdash \bot\ \mathsf{wf}}(\mathsf{P}_3) \qquad \frac{\Sigma \vdash P_1\ \mathsf{wf} \quad \Sigma \vdash P_2\ \mathsf{wf}}{\Sigma \vdash P_1 \wedge P_2\ \mathsf{wf}}(\mathsf{P}_4)$$

$$\frac{\Sigma \vdash P_1\ \mathsf{wf} \quad \Sigma \vdash P_2\ \mathsf{wf}}{\Sigma \vdash P_1 \vee P_2\ \mathsf{wf}}(\mathsf{P}_5) \qquad \frac{\Sigma \vdash P\ \mathsf{wf}}{\Sigma \vdash \neg P\ \mathsf{wf}}(\mathsf{P}_6)$$

$$\frac{\Sigma \vdash P_1\ \mathsf{wf} \quad \Sigma \vdash P_2\ \mathsf{wf}}{\Sigma \vdash P_1 \Rightarrow P_2\ \mathsf{wf}}(\mathsf{P}_7)$$

$$\frac{\Sigma, x : \tau \vdash P\ \mathsf{wf}}{\Sigma \vdash \forall x : \tau.\, P\ \mathsf{wf}}(\mathsf{P}_8) \qquad \frac{\Sigma, x : \tau \vdash P\ \mathsf{wf}}{\Sigma \vdash \exists x : \tau.\, P\ \mathsf{wf}}(\mathsf{P}_8)$$

Well-formed theories ($\vdash \Sigma\ \mathsf{wf}$):

$$\frac{}{\vdash \emptyset\ \mathsf{wf}}(\mathsf{Th}_1) \qquad \frac{\texttt{type}\ s \notin \Sigma}{\vdash \Sigma, \texttt{type}\ s[n]\ \mathsf{wf}}(\mathsf{Th}_2) \qquad \frac{x \notin \Sigma \quad \Sigma \vdash \tau\ \mathsf{wf}}{\vdash \Sigma, x : \tau\ \mathsf{wf}}(\mathsf{Th}_3)$$

$$\frac{\texttt{fun}\ f \notin \Sigma \quad \forall i,\ \Sigma \vdash \tau_i\ \mathsf{wf}}{\vdash \Sigma, \texttt{fun}\ f : \forall \boldsymbol{\alpha}.\, \tau_1, \ldots, \tau_n \to \tau_{n+1}\ \mathsf{wf}}(\mathsf{Th}_4)$$

$$\frac{\texttt{pred}\ p \notin \Sigma \quad \forall i,\ \Sigma \vdash \tau_i\ \mathsf{wf}}{\vdash \Sigma, \texttt{pred}\ p : \forall \boldsymbol{\alpha}.\, \tau_1, \ldots, \tau_n\ \mathsf{wf}}(\mathsf{Th}_5) \qquad \frac{\Sigma \vdash P\ \mathsf{wf}}{\vdash \Sigma, \texttt{axiom}\ \forall \boldsymbol{\alpha}.\, P\ \mathsf{wf}}(\mathsf{Th}_6)$$

Natural deduction rules ($\Sigma \models P$). For the sake of clarity, we write Σ, P for $\Sigma, \texttt{axiom}\ P$ in the following. A substitution σ over from type variables to types in extended to terms and predicates in the obvious way. We write $P[t/x]$ for the substitution of all the occurrences of a free variable x in P by a term t.

$$\frac{\texttt{axiom}\ \forall \boldsymbol{\alpha}.\, P \in \Sigma \quad \mathsf{Subst}(\sigma, \Sigma)}{\Sigma \models \sigma(P)}(\mathsf{Ax}) \qquad \frac{\Sigma \models Q \quad \Sigma, Q \models P}{\Sigma \models P}(\mathsf{Cut})$$

$$\frac{}{\Sigma \models \top}(\mathsf{True}) \qquad \frac{\Sigma \models \bot \quad \Sigma \vdash P\ \mathsf{wf}}{\Sigma \models P}(\mathsf{False}) \qquad \frac{\Sigma \vdash P\ \mathsf{wf}}{\Sigma \models P \vee \neg P}(\mathsf{EM})$$

$$\frac{\Sigma \models P \quad \Sigma \models Q}{\Sigma \models P \wedge Q}(\mathsf{And}_1) \qquad \frac{\Sigma \models P \wedge Q}{\Sigma \models P}(\mathsf{And}_2) \qquad \frac{\Sigma \models P \wedge Q}{\Sigma \models Q}(\mathsf{And}_3)$$

$$\frac{\Sigma \models P \quad \Sigma \vdash Q\ \mathsf{wf}}{\Sigma \models P \vee Q}(\mathsf{Or}_1) \qquad \frac{\Sigma \models Q \quad \Sigma \vdash P\ \mathsf{wf}}{\Sigma \models P \vee Q}(\mathsf{Or}_2)$$

$$\frac{\Sigma \models P \vee Q \quad \Sigma, P \models R \quad \Sigma, Q \models R}{\Sigma \models R}(\mathsf{Or}_3)$$

$$\frac{\Sigma \vdash P\ \mathsf{wf} \quad \Sigma, P \models \bot}{\Sigma \models \neg P}(\mathsf{Not}_1) \qquad \frac{\Sigma \models P \quad \Sigma \models \neg P}{\Sigma \models \bot}(\mathsf{Not}_2)$$

$$\frac{\Sigma \vdash P\ \mathsf{wf} \quad \Sigma, P \models Q}{\Sigma \models P \Rightarrow Q}(\mathsf{Imp}_1) \qquad \frac{\Sigma \models P \Rightarrow Q \quad \Sigma \models P}{\Sigma \models Q}(\mathsf{Imp}_2)$$

$$\frac{x \notin \Sigma \quad \Sigma \vdash \tau\ \mathsf{wf} \quad \Sigma, x : \tau \models P}{\Sigma \models \forall x : \tau.\, P}(\mathsf{Forall}_1)$$

$$\frac{\Sigma \models \forall x : \tau.\, P \quad \Sigma \vdash t : \tau}{\Sigma \models P[t/x]}(\mathsf{Forall}_2)$$

$$\frac{\Sigma \vdash t : \tau \quad x \notin \Sigma \quad \Sigma \models P[t/x]}{\Sigma \models \exists x : \tau.\, P}(\mathsf{Exists}_1)$$

$$\frac{\Sigma \models \exists x : \tau.\, P \quad x \notin \Sigma \quad \Sigma, x : \tau, P \models Q}{\Sigma \models Q}(\mathsf{Exists}_2)$$

Deduction rules for equality:

$$\frac{\Sigma \vdash t : \tau}{\Sigma \models t = t}(\mathsf{Eq}_1) \qquad \frac{\Sigma \models x = y \quad \Sigma, z : \tau \vdash P\ \mathsf{wf} \quad \Sigma \models P[x/z]}{\Sigma \models P[y/z]}(\mathsf{Eq}_2)$$

Arithmetic: An arithmetic proposition P is a proposition built from variables of type *int*, integers constants, the functions symbols *add* and *sub*, the predicates *lt, le, gt, ge* and the equality. If $x_1, \ldots, x_n$ are the free variables of P, we write $x_1, \ldots, x_n \models_A P$ whenever P is valid (it is decidable; see for instance the Omega test [20]). Then we have the following deduction rule for arithmetic:

$$\frac{x_1, \ldots, x_n \models_A P}{\Sigma \models \forall x_1 : \mathit{int}.\, \ldots \forall x_n : \mathit{int}.\, P}(\mathsf{Arith})$$