Reflecting Proofs in First-Order Logic with Equality ⋆

Evelyne Contejean and Pierre Corbineau

PCRI — LRI (CNRS UMR 8623) & INRIA Futurs
Bât. 490, Université Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France

Abstract. Our general goal is to provide better automation in interactive proof assistants such as Coq. We present an interpreter of proof traces in first-order multi-sorted logic with equality. Thanks to the reflection ability of Coq, this interpreter is both implemented and formally proved sound — with respect to a reflective interpretation of formulae as Coq properties — inside Coq's type theory. Our generic framework allows to interpret proofs traces computed by any automated theorem prover, as long as they are precise enough: we illustrate that on traces produced by the CiME tool when solving unifiability problems by ordered completion. We discuss some benchmark results obtained on the TPTP library.

The aim of this paper is twofold: first we want to validate a reflective approach for proofs in interactive proof assistants, and second show how to provide a better automation for such assistants. Both aspects can be achieved by using external provers designed to automatically solve some problems of interest: these provers can "feed" the assistant with large proofs, and help to compare the direct and the reflective approaches, and they can also release the user from (parts of) the proof.

The proof assistant doesn't rely on the soundness of the external tool, but keeps a skeptical attitude towards the external traces by rechecking them. Moreover incompleteness of this tool is not an issue either, since when it fails to produce an answer, the user simply has to find another way to do his proof. But a key point is that it has to produce a trace which can be turned into a proof.

Proof checkers usually have a very fine grained proof notion, whereas automated theorem provers tend to do complex inferences such as term normalization and paramodulation in one single step. Reflection techniques [10] provide a good intermediate layer to turn traces missing a lot of implicit information into fully explicit proofs. They rely on the computation abilities of the proof assistant for trivial parts of proofs, leaving the hard but interesting work of finding proofs to automated tools. Bezem *et al.* [3] use reflection techniques to handle the clausification part of a proof but the derivation of the empty clause is provided by an external tool.

Our approach extends the reflection technique to the proof itself, turning the proof assistant into a *skeptical trace interpreter* from an intermediate language of proof traces to its own native format. We have implemented this technique inside the Coq proof assistant [17] using a sequent calculus for multi-sorted intuitionistic first-order logic with equality as a semantics for the intermediate language. To validate the reflective approach, we used the CiME tool [4] to produce traces when solving word and unifiability problems by ordered completion and had these traces checked by Coq.

⋆ Supported by the EU coordinated action TYPES and the French RNTL project AVERROES
`www.cs.chalmers.se/Cs/Research/Logic/Types/`
`www-verimag.imag.fr/AVERROES/`

Other works integrate either reflection and/or rewriting inside Coq. Nguyen [14] explains how to produce term rewriting proofs, but does not use reflection, whereas Alvarado [1] provides a reflection framework dedicated to proofs of equality of terms. Both of them consider that the rewriting system is fixed *a priori*. Our approach is close to the work of Crégut [6] who also interprets proof traces thanks to reflection, but for quantifier-free formulae in Peano's Arithmetic.

In our work, the rewriting system changes during the completion process, and in order to address the problem of the trace, we add some information on the usual rules of ordered completion. Thanks to this extra information, only the useful rules are extracted from the completion process, and the resulting formal proof is significantly shorter than the completion process. But this is far from a formal proof. We explain how to turn the annotations into useful lemmata, that is universally quantified equalities together with their proofs as sequences of equational steps.

Section 1 presents a general framework for reflection of first-order logic with equality in type theory. In Section 2, we give key details of the implementation. In Section 3, we briefly recall ordered completion, give an annotated version of the completion rules and show how to extract a CiME proof from a successful completion run. In Section 4, we explain the translation of CiME proofs into Coq reified proofs. In Section 5, we comment some benchmarks obtained on the TPTP library with CiME and Coq.

1 Type Theory and the Reflection Principle

1.1 Type Theory and the Conversion Rule

Coq's type theory is an extension of dependently-typed λ-calculus with inductive types, pattern matching and primitive recursion. Coq acts as an interpreter/type-checker for this language. The use of the *proofs as programs* paradigm allows its use as an interactive proof assistant: the typing relation $\Gamma \vdash t : T$ can be seen either as *"according to the types of variables in Γ, t is an object in set T"* or as *"supposing the hypotheses in Γ, t is a proof of the assertion T"*. Together with the λ function binder, Coq's most important construction is the $\forall$ binding operator, which builds either a dependent function type or a universally quantified logical proposition. The term $\forall x : A.B$ is written $A \rightarrow B$ as long as x does not occur free in B; in this case it represents the function space from type A to type B or the implication between propositions A and B. To establish a clear distinction between informative objects (datatypes and functions) and non-informative objects (proofs of propositions), types intended as datatypes are in the sort Set and logic propositions are in the sort Prop.

A main feature of Coq's type theory, which will be essential for the reflection mechanism, is the *conversion* rule: if $\Gamma \vdash t : T$ then $\Gamma \vdash t : T'$ for any type T' equivalent to T, i.e. having the same normal form with respect to the reduction of Coq terms. In particular, any term t can be proved equal to its normal form $\mathrm{nf}(t)$ inside Coq's type theory : we can use the reflexivity axiom $\mathtt{refl}_=$ to build the proof $(\mathtt{refl}_=\ t) : t = t$ and thanks to the conversion rule this is also a proof of $t = \mathrm{nf}(t)$ *e.g.* take the term 2×2 which reduces to 4, then $(\mathtt{refl}_=\ 4)$ is a proof of $4 = 4$ and a proof of $2 \times 2 = 4$ (both propositions have $4 = 4$ as normal form).

1.2 The Reflection Principle

The power of type theory appears when building types or propositions by case analysis on informative objects: suppose there is a type `form` of sort Set intended as a concrete representation of logical formulas, and an interpretation function $[\![\_]\!] : \texttt{form} \rightarrow \mathsf{Prop}$, then one can define a function $\texttt{decide} : \texttt{form} \rightarrow \texttt{bool}$ deciding whether an object in `form` represents a tautology, and prove a correctness theorem:

$$\texttt{decide\_correct} : \forall F : \texttt{form}, ((\texttt{decide}\ F) = \texttt{true}) \rightarrow [\![F]\!]$$

This gives a useful way of proving that some proposition A in the range of $[\![\_]\!]$ is valid. First, by some external means, A is *reified*, i.e. some representation $\dot{A}$ such that $[\![\dot{A}]\!]$ is convertible to A is computed. Then the term $(\texttt{decide\_correct}\,\dot{A}\,(\texttt{refl}_{=}\texttt{true}))$ is built. This term is well typed if, and only if $(\texttt{decide}\,\dot{A})$ is convertible to `true`, and then its type is $[\![\dot{A}]\!]$ so this term is a proof of A. Otherwise, the term is not typable. The advantage of this approach is that it is left to the internal reduction procedure of Coq to check if A is provable. Moreover, careful implementation of the `decide` function allows some improvement in the time and space required to check the proof.

The distinction between properties and their representations is necessary since there is no way to actually compute something from a property (for example there is no function of type $\mathsf{Prop} \rightarrow \mathsf{Prop} \rightarrow \texttt{bool}$ deciding the syntactic equality) since there is no such thing as pattern-matching or primitive recursion in Prop, whereas the representation of properties can be defined as an inductive type such as `form` on which functions can be defined by case analysis.

As tempting as this approach may seem, there are several pitfalls to avoid in order to be powerful enough. First, the propositions that can be proven thanks to this approach are the interpretations of the representations $F : \texttt{form}$ such that $(\texttt{decide}\,F) = \texttt{true}$. This set is limited by the range of the interpretation function and by the power of the decision function: the function that always yields `false` is easily proved correct but is useless. Besides, we need to take into account space and time limitations induced by the cost of the reduction of complex Coq terms inside the kernel.

1.3 Reflecting First-Order Proofs

Since validity in first-order logic is undecidable, we have two options if we want to prove first-order assertions by reflection: either restrict the `decide` function to a decidable fragment of the logic, or change the `decide` function into a proof-checking function taking a proof trace (of type $\texttt{proof} : \mathsf{Set}$) as an extra argument. The correctness theorem for this function `check` reads :

$$\texttt{check\_correct} : \forall \pi : \texttt{proof}.\forall F : \texttt{form}.((\texttt{check}\,F\,\pi) = \texttt{true}) \rightarrow [\![F]\!]$$

The proof process using this proof trace approach is shown in Figure 1, it shows how an external tool can be called by Coq to compute a proof trace for a given formula. The reification phase is done inside the Coq system at the ML level and not by a type-theory function (remember there is no case analysis on Prop).

The proof traces can have very different forms, the key point being the amount of implicit information that `check` will have to recompute when the kernel will type-check the proof. Moreover, if too many things have to be recalculated, the correctness theorem may become trickier to prove. What we propose is to give a derivation tree for a sequent calculus as a proof trace, and this is what the next section is about.

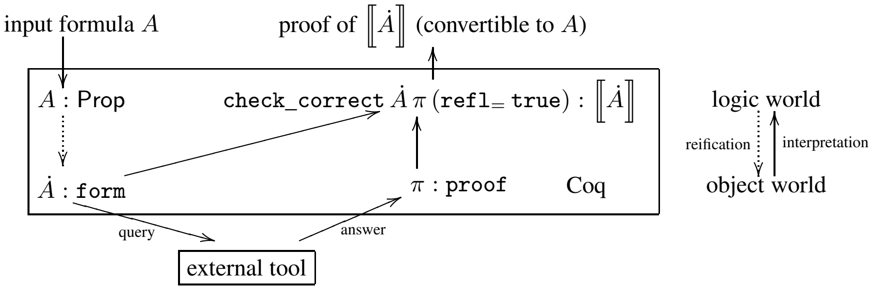

Fig. 1. Reflection scheme with proof traces

2 Proof Reflection for Multi-Sorted First-Order Logic

2.1 Representation for Indexed Collections

The first attempt to represent collections of indexable objects with basic lists indexed by unary integers as in [3] was most inefficient, consuming uselessly time and space resources. For efficiency purposes, lists were replaced by binary trees indexed by binary integers $\mathbb{B}$. These binary trees $\mathbb{T}(\tau)$ containing objects of type τ are equipped with access and insertion functions $\texttt{get} : \mathbb{T}(\tau) \to \mathbb{B} \to \tau^?$ and $\texttt{add} : \mathbb{T}(\tau) \to \mathbb{B} \to \tau \to \mathbb{T}(\tau)$.

The question mark in the `get` function stands for the basic `option` type constructor adding a dummy element to any type. Indeed any Coq function must be total, which means that even the interpretation of badly formed objects needs a value. Most of the time the `option` types will be the solution to represent partiality.

Trees will sometimes be used as indexable stacks by constructing a pair in type $\mathbb{B} \times \mathbb{T}(\tau)$, the first member being the next free index in the second member. The empty stack and the empty tree will be denoted $\emptyset$. As a shortcut, if $S = (i, T)$, the notation $S; x$ stands for $(i + 1, \texttt{add}(T, i, x))$ (pushing x on top of S). The i index in $S; i : x$ means that i is the stack pointer of S, i.e. the index pointing to x in $S; i : x$ (see Example 1). The notation S_j stands for $\texttt{get}(T, j)$, assuming j is a valid index, i.e. $j < i$.

2.2 Representation and Interpretation of Formulae

Since Coq's quantifiers are typed, in order to be able to work with problems involving several types that are unknown when proving the reflection theorems, our representation will be *parameterized* by a *domain signature*. This enables the representation of multi-sorted problems, and not only problems expressed with a domain chosen *a priori*. We represent the quantification domains by our binary integers, referring to a binary tree $\mathrm{Dom} : \mathbb{T}(\mathsf{Set})$ which we call *domain signature*. The interpretation function, given a domain signature, is a Coq function $\llbracket \_ \rrbracket_{\mathrm{Dom}} : \mathbb{B} \to \mathsf{Set}$, which maps undefined indices to $\mathbb{1} : \mathsf{Set}$, the set with one element.

Given a domain signature, function symbols are defined by a dependent record: its first field is a (possibly empty) list of binary integers $d_1, \ldots, d_n$, the second field is an integer r for the range of the function, and the third field is the function itself, of type $\llbracket d_1 \rrbracket_{\mathrm{Dom}} \to \ldots \to \llbracket d_n \rrbracket_{\mathrm{Dom}} \to \llbracket r \rrbracket_{\mathrm{Dom}}$ (its type is computed from Dom and the two first fields). Predicate symbols are defined in a similar way, except there is no range field and the last field has type $\llbracket d_1 \rrbracket_{\mathrm{Dom}} \to \ldots \to \llbracket d_n \rrbracket_{\mathrm{Dom}} \to \mathsf{Prop}$. The *function signature* Fn is defined as a tree of function symbols, and the *predicate signature* Pred is defined as a tree of predicate symbols.

Example 1. In order to express properties of integers in $\mathbb{N}$ and arrays of integers (in $\mathcal{A}_{\mathbb{N}}$), we define the domain signature as $\mathrm{Dom} = 1 : \mathbb{N}\,; 2 : \mathcal{A}_{\mathbb{N}}$.

In the signature, we put the $\mathbf{0}$ constant as well as two relations $<$ and $\geq$ over the integers. We also add the `get` and `set` functions over arrays and the `sorted` predicate. The `valid` predicate states that an integer is a valid index for a given array. The function and predicate signatures corresponding to this theory will be:

$$\mathrm{Fn} = 1 : \{\emptyset, 1, \mathbf{0}\}; 2 : \{[2; 1], 1, \mathtt{get}\}; 3 : \{[2; 1; 1], 2, \mathtt{set}\}$$

$$\mathrm{Pred} = 1 : \{[1; 1], <\}; 2 : \{[1; 1], \geq\}; 3 : \{[2], \mathtt{sorted}\}; 4 : \{[1; 2], \mathtt{valid}\}$$

Definition 1 (term, formula). *Terms and formulae are recursively defined as follows:*

$$\mathtt{term} := \mathrm{Fv}_{\mathbb{B}} \mid \mathrm{Bv}_{\mathbb{N}} \mid \mathrm{App}\ \mathbb{B}\ \ \mathtt{args} \qquad\qquad \mathtt{args} := \emptyset \mid \mathtt{term}, \mathtt{args}$$

$$\mathtt{form} := \mathrm{Atom}\ \mathbb{B}\ \ \mathtt{args} \mid \mathtt{term} \underset{\mathbb{B}}{\dot{=}} \mathtt{term} \mid \dot{\bot} \mid \dot{\forall}_{\mathbb{B}}\, \mathtt{form} \mid \dot{\exists}_{\mathbb{B}}\, \mathtt{form}$$
$$\mid \mathtt{form} \dot{\rightarrow} \mathtt{form} \mid \mathtt{form} \dot{\wedge} \mathtt{form} \mid \mathtt{form} \dot{\vee} \mathtt{form}$$

In terms, the Fv constructor represents free variables and their indices will refer to the slot they use in the sequent context (see below). The Bv constructor represents bound variables under quantifiers, using the deBruijn notation [7]: the indices are unary integers $\mathbb{N}$, 0 standing for the variable bound by the innermost quantifier over the position of the variable, 1 for the next innermost, etc. The indices in the App and Atom constructors refer to symbols in the signature, whereas those in the equality and quantifiers refer to domains in Dom. There is no variable in the quantifiers since the deBruijn notation takes care of which quantifier binds which variable without having to name the variable. The logical negation of a formula F can be expressed by $F \dot{\rightarrow} \dot{\bot}$.

A term is closed if it contains no Bv constructor, and a formula is closed if all Bv constructors have indices less than their quantifier depth.

Example 2. Using the signature of the previous example, the property:

$\forall a : \mathcal{A}_{\mathbb{N}}, \mathtt{sorted}(a) \rightarrow \exists i : \mathbb{N}, \forall : j : \mathbb{N}, \mathtt{valid}(j, a) \rightarrow$
$((j < i) \rightarrow (\mathtt{get}(a, j) < \mathbf{0})) \wedge ((j \geq i) \rightarrow (\mathtt{get}(a, j) \geq \mathbf{0}))$ is represented by:

$$\dot{\forall}_2(\mathrm{Atom}\ 3\ \mathrm{Bv}_0) \dot{\rightarrow} \dot{\exists}_1 \dot{\forall}_1 (\mathrm{Atom}\ 4\ (\mathrm{Bv}_0, \mathrm{Bv}_2)) \dot{\rightarrow}$$
$$(\mathrm{Atom}\ 1\ (\mathrm{Bv}_0, \mathrm{Bv}_1)) \dot{\rightarrow} (\mathrm{Atom}\ 1\ (\mathrm{App}\ 2\ (\mathrm{Bv}_2, \mathrm{Bv}_0), \mathrm{App}\ 1\ \emptyset)) \dot{\wedge}$$
$$(\mathrm{Atom}\ 2\ (\mathrm{Bv}_0, \mathrm{Bv}_1)) \dot{\rightarrow} (\mathrm{Atom}\ 2\ (\mathrm{App}\ 2\ (\mathrm{Bv}_2, \mathrm{Bv}_0), \mathrm{App}\ 1\ \emptyset))$$

Definition 2 (Sequent). *A sequent is a pair written $\Gamma \vdash G$, where Γ is the context, of type $\mathbb{T}(\mathbb{B} + \mathtt{form})$ containing objects in $\mathbb{B}$ or in* `form`*. Objects in* `form` *represent logical hypotheses whereas objects in $\mathbb{B}$ represent the domain of assumed variables (referred to using the Fv constructor). $G : \mathtt{form}$ is called the goal of the sequent.*

On Figure 2, we explain how to interpret reified objects as Coq propositions. The interpretation functions use *global and local valuations*, these valuations contain dependent pairs $\{v \in d\}$ in type $\mathtt{val} : \mathrm{Set}$. They contain a domain index d and an object $v : [\![d]\!]_{\mathrm{Dom}}$. The letter γ will usually denote global valuations, in $\mathbb{T}(\mathtt{val})$. We say γ is an *instantiation* for a context Γ if it maps indices of global variables in Γ to pairs in the same domain (it gives values to global variables in Γ). The local context usually written δ, is a list of the above pairs (addition of x in the list L is written $(x :: L)$).

The interpretation of terms takes an extra argument which is the intended domain r of the result, and returns an optional value in $[\![r]\!]^{?}_{\mathrm{Dom}}$. If the represented term is not well typed, the dummy optional value is returned. The interpreter for arguments uses an accumulator Φ which is successively applied to the interpretation of arguments. This

Terms	$\llbracket \mathrm{Bv}_n \rrbracket^{r}_{\gamma,\delta} = x$ if $\delta_n = \{x \in r'\}$ and $r = r'$
	$\llbracket \mathrm{Fv}_i \rrbracket^{r}_{\gamma,\delta} = x$ if $\gamma_i = \{x \in r'\}$ and $r = r'$
	$\llbracket \mathrm{App}\ i\ a \rrbracket^{r}_{\gamma,\delta} = \llbracket f \mid a \rrbracket^{d}_{\gamma,\delta}$ if $\mathrm{Fn}_i = (d, r', f)$ and $r = r'$
Arguments	$\llbracket \Phi \mid \emptyset \rrbracket^{\emptyset}_{\gamma,\delta} = \Phi \qquad \llbracket \Phi \mid t :: a \rrbracket^{r::d}_{\gamma,\delta} = \left\llbracket \Phi(\llbracket t \rrbracket^{r}_{\gamma,\delta}) \middle\vert\, a \right\rrbracket^{d}_{\gamma,\delta}$
Formulae	$\llbracket \mathrm{Atom}\ i\ a \rrbracket_{\gamma,\delta} = \llbracket P \mid a \rrbracket^{d}_{\gamma,\delta}$ if $\mathrm{Pred}_i = (d, P)$
	$\left\llbracket t_1 \underset{d}{\dot{=}} t_2 \right\rrbracket_{\gamma,\delta} = \llbracket t_1 \rrbracket^{d}_{\gamma,\delta} \underset{\llbracket d \rrbracket_{\mathrm{Dom}}}{=} \llbracket t_2 \rrbracket^{d}_{\gamma,\delta} \qquad \left\llbracket \dot{\bot} \right\rrbracket_{\gamma,\delta} = \bot$
	$\left\llbracket \dot{\forall}_d F \right\rrbracket_{\gamma,\delta} = \forall x : \llbracket d \rrbracket_{\mathrm{Dom}}, \llbracket F \rrbracket_{\gamma, \{x \in d\} :: \delta}$
	$\left\llbracket \dot{\exists}_d F \right\rrbracket_{\gamma,\delta} = \exists x : \llbracket d \rrbracket_{\mathrm{Dom}}, \llbracket F \rrbracket_{\gamma, \{x \in d\} :: \delta}$
	$\llbracket F_1 \dot{\rightarrow} F_2 \rrbracket_{\gamma,\delta} = \llbracket F_1 \rrbracket_{\gamma,\delta} \rightarrow \llbracket F_2 \rrbracket_{\gamma,\delta}$
	$\llbracket F_1 \dot{\wedge} F_2 \rrbracket_{\gamma,\delta} = \llbracket F_1 \rrbracket_{\gamma,\delta} \wedge \llbracket F_2 \rrbracket_{\gamma,\delta}$
	$\llbracket F_1 \dot{\vee} F_2 \rrbracket_{\gamma,\delta} = \llbracket F_1 \rrbracket_{\gamma,\delta} \vee \llbracket F_2 \rrbracket_{\gamma,\delta}$
Contexts	$\llbracket \emptyset \Vert \Psi \rrbracket = \Psi(\emptyset) \qquad \llbracket \Gamma ; F \Vert \Psi \rrbracket = \left\llbracket \Gamma \middle\Vert \gamma \mapsto \llbracket F \rrbracket_{\gamma,\emptyset} \rightarrow \Psi(\gamma) \right\rrbracket$
	$\llbracket \Gamma ; id : \Vert \Psi \rrbracket = \left\llbracket \Gamma \middle\Vert \gamma \mapsto \forall x : \llbracket d \rrbracket_{\mathrm{Dom}}, \Psi(\mathtt{add}\ \gamma\ i\ \{x \in d\}) \right\rrbracket$
Sequents	$\llbracket \Gamma \vdash G \rrbracket = \left\llbracket \Gamma \middle\Vert \gamma \mapsto \llbracket G \rrbracket_{\gamma,\emptyset} \right\rrbracket$

Fig. 2. Interpretation of terms, formulae, sequents

accumulator trick is necessary to build well typed terms, as well as the use of a Coq proof of $r = r'$ to coerce objects of $\llbracket r' \rrbracket_{\mathrm{Dom}}$ to $\llbracket r \rrbracket_{\mathrm{Dom}}$. The interpretation of atomic formulae uses the arguments interpreter with a different kind of accumulator. The interpretation of badly formed formulae is $\top$, the trivial formula. The interpretation of contexts is presented inside out so it is easier to reason about, it builds a function Ψ from global valuations to Prop by adding hypotheses recursively.

2.3 Proof traces

We define the proof traces and their meaning in Figure 3. The judgement $\pi : \Gamma \vdash G$ means *"π is a correct proof trace for the sequent $\Gamma \vdash G$"*. We adapted Roy Dychkoff's contraction free sequent calculus for intuitionistic logic [8], in order to allow multiple sorts and possibly empty domains. Even though the choice of this sequent calculus may seem exotic, the proof technique is quite generic and our proofs may be easily adapted to other kinds of mono-succedent sequent calculi such as classical ones with excluded middle axiom or implicit non-empty domains.

In order to reify first-order reasoning, it is necessary to decide syntactic equality between terms and between formulae, which is done by the boolean functions $=_{\mathtt{term}}$ and $=_{\mathtt{form}}$. The `inst` function implements the substitution of a given closed term (no lift operator needed) to the first bound variable in a formula, which is used in the definition of proof steps for quantifiers. Finally, the `rewrite` function replaces a closed term by another at a given set of occurrences, checking that the first term is the same as the subterms at the rewrite positions.

2.4 Correctness proof

For every object defined in the last paragraphs (terms, arguments, formulae, contexts, sequents), there is an implicit notion of well-formedness contained in the definition of the interpretation functions. These well-formedness properties are implemented inside Coq both as inductive predicates in Prop named `WF_`$*$ (with $* =$ `term`, `args`, `form`...) and as boolean functions `check_`$*$. For every object a correctness lemma is proved:

$$\texttt{WF\_checked\_}* : \forall x : *, (\texttt{check\_}*\ x) = \texttt{true} \to (\texttt{WF\_}*\ x)$$

The next step is the definition of `WF_proof` which is a Coq inductive predicate representing the ":" in the well-formedness of proofs. Then, a `check_proof` boolean function is implemented and the corresponding `WF_checked_proof` lemma can be proved. Using these, the fundamental theorem can be stated.

Theorem 1 (Logical soundness). *There are Coq proof terms of these two theorems:*

$$\forall \pi : \texttt{proof}. \forall \Gamma, \forall G, (\texttt{WF\_sequent}\ \Gamma \vdash G) \to (\texttt{WF\_proof}\ \Gamma \vdash G\ \pi) \to [\![\Gamma \vdash G]\!] \tag{1}$$

$$\begin{aligned} \forall \pi : \texttt{proof}. \forall \Gamma, \forall G, &(\texttt{check\_sequent}\ \Gamma \vdash G) = \texttt{true} \to \\ &(\texttt{check\_proof}\ \Gamma \vdash G\ \pi) = \texttt{true} \to [\![\Gamma \vdash G]\!] \end{aligned} \tag{2}$$

$$\dfrac{}{(\mathrm{Ax}\ i) : \Gamma \vdash G}\ \Gamma_i = G \qquad \dfrac{\pi_1 : \Gamma \vdash A \quad \pi_2 : \Gamma; A \vdash G}{(\mathrm{Cut}\ A\ \pi_1\ \pi_2) : \Gamma \vdash G} \qquad \dfrac{}{(\bot_E\ i) : \Gamma \vdash G}\ \Gamma_i = \dot{\bot}$$

$$\dfrac{\pi : \Gamma; A \vdash B}{(\to_I\ \pi) : \Gamma \vdash A \dot{\to} B} \qquad \dfrac{\pi : \Gamma; B \vdash G}{(\to_E\ i\ j\ \pi) : \Gamma \vdash G} \begin{cases} \Gamma_i = A \dot{\to} B \\ \Gamma_j = A \end{cases}$$

$$\dfrac{\pi_1 : \Gamma; A; B \dot{\to} C \vdash B \quad \pi_2 : \Gamma; C \vdash G}{(\to_D\ i\ \pi_1\ \pi_2) : \Gamma \vdash G}\ \Gamma_i = (A \dot{\to} B) \dot{\to} C$$

$$\dfrac{\pi_1 : \Gamma \vdash A \quad \pi_2 : \Gamma \vdash B}{(\wedge_I\ \pi_1\ \pi_2) : \Gamma \vdash A \dot{\wedge} B} \qquad \dfrac{\pi : \Gamma; A; B \vdash G}{(\wedge_E\ i\ \pi) : \Gamma \vdash G}\ \Gamma_i = A \dot{\wedge} B$$

$$\dfrac{\pi : \Gamma; A \dot{\to} B \dot{\to} C \vdash G}{(\wedge_D\ i\ \pi) : \Gamma \vdash G}\ \Gamma_i = (A \dot{\wedge} B) \dot{\to} C$$

$$\dfrac{\pi : \Gamma \vdash A}{(\vee_{I_1}\ \pi) : \Gamma \vdash A \dot{\vee} B} \qquad \dfrac{\pi : \Gamma \vdash B}{(\vee_{I_2}\ \pi) : \Gamma \vdash A \dot{\vee} B} \qquad \dfrac{\pi_1 : \Gamma; A \vdash G \quad \pi_2 : \Gamma; B \vdash G}{(\vee_E\ i\ \pi_1\ \pi_2) : \Gamma \vdash G}\ \Gamma_i = A \dot{\vee} B$$

$$\dfrac{\pi : \Gamma; A \dot{\to} C; B \dot{\to} C \vdash G}{(\vee_D\ i\ \pi) : \Gamma \vdash G}\ \Gamma_i = (A \dot{\vee} B) \dot{\to} C$$

$$\dfrac{\pi : \Gamma;_j d \vdash (\texttt{inst}\ A\ \mathrm{Fv}_j)}{(\forall_I\ \pi) : \Gamma \vdash \dot{\forall}_d A} \qquad \dfrac{\pi : \Gamma; (\texttt{inst}\ A\ t) \vdash G}{(\forall_E\ i\ t\ \pi) : \Gamma \vdash G}\ \Gamma_i = \dot{\forall}_d A$$

$$\dfrac{\pi_1 : \Gamma \vdash \dot{\forall}_d A \quad \pi_2 : \Gamma; B \vdash G}{(\forall_D\ i\ \pi_1\ \pi_2) : \Gamma \vdash G}\ \Gamma_i = (\dot{\forall}_d A) \dot{\to} B$$

$$\dfrac{\pi : \Gamma \vdash (\texttt{inst}\ A\ t)}{(\exists_I\ t\ \pi) : \Gamma \vdash \dot{\exists}_d A} \qquad \dfrac{\pi : \Gamma;_j d; (\texttt{inst}\ A\ \mathrm{Fv}_j) \vdash G}{(\exists_E\ i\ \pi) : \Gamma \vdash G}\ \Gamma_i = \dot{\exists}_d A$$

$$\dfrac{\pi : \Gamma; \dot{\forall}_d (A \dot{\to} B) \vdash G}{(\exists_D\ i\ \pi) : \Gamma \vdash G}\ \Gamma_i = (\dot{\exists}_d A) \dot{\to} B$$

$$\dfrac{}{(=_I\ \ t) : \Gamma \vdash t \underset{\tau}{\doteq} t} \qquad \dfrac{\pi : \Gamma; A \vdash G}{(=_D\ \ i\ \pi) : \Gamma \vdash G}\ \Gamma_i = (t \underset{\tau}{\doteq} t) \dot{\to} A$$

$$\dfrac{\pi : \Gamma \vdash (\texttt{rewrite}\ s\ t\ p\ G)}{(=_{E_1}\ \ i\ \to\ p\ \pi) : \Gamma \vdash G}\ \Gamma_i = s \underset{d}{\doteq} t \qquad \dfrac{\pi : \Gamma \vdash (\texttt{rewrite}\ t\ s\ p\ G)}{(=_{E_1}\ \ i\ \leftarrow\ p\ \pi) : \Gamma \vdash G}\ \Gamma_i = s \underset{d}{\doteq} t$$

$$\dfrac{\pi : \Gamma; (\texttt{rewrite}\ s\ t\ p\ A) \vdash G}{(=_{E_2}\ \ i\ j\ \to\ p\ \pi) : \Gamma \vdash G} \begin{cases} \Gamma_i = s \underset{d}{\doteq} t \\ \Gamma_j = A \end{cases} \qquad \dfrac{\pi : \Gamma; (\texttt{rewrite}\ t\ s\ p\ A) \vdash G}{(=_{E_2}\ \ i\ j\ \leftarrow\ p\ \pi) : \Gamma \vdash G} \begin{cases} \Gamma_i = s \underset{d}{\doteq} t \\ \Gamma_j = A \end{cases}$$

Fig. 3. Well-formedness of proofs

Coq proofs for these theorems, along with all the definitions above are available at `http://www.lri.fr/~corbinea/ftp/programs/rfo.tar.gz`.

Sketch of the proof. (2) is a consequence of (1) by composition with the `WF_checked_*` lemmata. Formula (1) is proved by induction on the proof trace, so it amounts to prove that in each step the interpretation of the conclusion is a consequence of the interpretation of the premises. Many intermediate steps are needed, the key ones being lemmata about the semantics of `inst` and `rewrite`, weakening lemmata, and lemmata about the stability of the interpretation functions (an interpretation is preserved by the addition of variables in the context).

3 Rewriting Traces for Ordered Completion

3.1 Completion Rules

The purpose of ordered completion [15, 13, 2] is to build a convergent rewriting system from a set of equations in order to decide the word problem. Some provers such as CiME [4] or Waldmeister [12] use an enhanced version of ordered completion in order to solve unifiability problems instead of word problems.

We adopt the classical presentation of the completion process as a sequence of applications of inference rules. The input is a pair $(E_0, s = t)$, where E_0 is a set of (implicitly universally quantified) equalities defining an equational theory, and $s = t$ is a conjecture, that is an implicitly existentially quantified equation. The output is True when there exists a substitution σ such that $s\sigma$ and $t\sigma$ are equal modulo E_0, False otherwise. Of course, this semi-decision procedure may not terminate. The whole procedure is parameterized by a reduction ordering $>$. The completion rules are working on a triple (E, R, C), where E and C are sets of equations (unordered pairs of terms), and R is a set of rules (ordered pairs of terms).

The set of completion rules may be divided into several subsets. First the rule **Init** initializes the process, by building the initial triple from the initial set of axioms and the conjecture. Then there are the rules **Orient** and **Orient'** which create rewrite rules for R from unordered pairs of terms $u = v$ in E. If u and v are comparable for $>$, a single rewrite rule is created, otherwise, two rules are created since it may be the case that $u\sigma > v\sigma$ or $v\sigma > u\sigma$, depending on σ. The rewrite rules are used by the **Rewrite**, **Rewrite'**, **Collapse** and **Compose** for rewriting respectively in an equation, in a conjecture, in the left-hand side of a rule and in the right-hand side of a rule. Some new facts are computed by deduction, either between two rewrite rules (**Critical pair**) or between a rewrite rule and a conjecture (**Narrow**).

A recursive application of the rules may run forever, or stop with an application of the **Success** rule. Due to lack of space, we do not recall the rules, they can be read from Figure 4 by erasing the annotations.

3.2 Completion Rules with Traces

In order to build a trace for a formal proof management system as Coq, the inference rules have to be annotated with some additional information. From this respect, there are two kinds of rules, those which *simplify* equations or rules by rewriting the inside terms, and those which *create* some new facts. The application of the simplification rules is recorded in the rewritten term itself as a part of its history whereas the new facts contain the step by which they were created as a trace. Moreover, one has to precisely identify the left- and right-hand sides of equations, which leads to duplicate some of the rules. We shall only write down the left version of them and mention that there is a right version when there is no ambiguity. Hence the data structures are enriched as follows:

- An equation is a pair of terms with a trace. It will sometimes be denoted by $u = v$, u and v being the two terms, when the trace is not relevant.
- A term u has a current version u^*, an original version u^0, and an history, that is sequence of rewriting steps, which enables to rewrite u^0 into u^*.
- A rewriting step is given by a position, a substitution and a rewrite rule.
- A rewrite rule is an oriented ($+$ or $-$) equation $u = v$ denoted by $u \xrightarrow{+} v$ or $v \xrightarrow{-} u$.
- A trace is either an axiom (ω), or a peak corresponding with a critical pair (κ) or with a narrowing (either in a left-hand side (ν_L) or in a right-hand side (ν_R)).
- A peak is given by its top term t, two rewrite rules rl_1, rl_2, and the position p where the lowest rewrite rule rl_2 has to be applied to t: ${}^{rl_2}_{p} \leftarrow t \rightarrow^{rl_1}_{\Lambda}$.

An axiom, that is a pair of usual terms $s = t$, is lifted into an equation $\widehat{s = t}$ by turning s and t into general terms with an empty history and by adding the trace $\omega(s = t)$. Given a term u, a rule $l \to r$, a position p and a substitution σ, when $u^*|_p = l^*\sigma$ the term u can be rewritten into a new term $Rew(u, \longrightarrow^{l \to r}_{p,\sigma})$ where the original version keeps the same, the current value is equal $u^*[r^*\sigma]_p$ and the new history is equal to $h@(p, \sigma, l \to r)$, h being the history of u. The annotated completion rules are given in Figure 4.

3.3 Constructing a Formal Proof from a Trace

When a triple (E, R, C) can trigger the completion rule **Success** all the needed information for building a formal proof can be extracted from the trace of the conjecture $s = t$ such that s^* and t^* are unifiable. Building the formal proof is done in two steps, first the information (as sequences of equational steps) is extracted from the trace, then the proof is built (either as a reified proof object or as a Coq script).

Word Problem We shall first discuss the easiest case, when the original conjecture is an equation between two closed terms. The narrowing rules never apply, the set of conjectures is always a singleton, and the **Success** rule is triggered when both current sides of the conjecture are syntactically equal.

There are two possibilities for building a proof. The first one is to give a sequence of equational steps with respect to E_0 between both sides of the conjecture (cut-free proof). Since extracting the information from the trace in order to build such a proof is quite involved (the critical pairs have to be recursively unfolded), we start by the other alternative which is more simpler, where the proof is given by a list of lemmata mirroring the computation of critical pairs.

Critical Pairs as Cuts First, it is worth noticing that all the applications of completion rules made so far do not have to be considered, but only the useful ones, which can be extracted and sorted by a dependency analysis, starting from the rewrite rules which occur in the history of s and t.

$$\textbf{Init}\ \frac{}{\widehat{E_0}, \emptyset, \{\widehat{s = t}\}}\ \text{ if } E_0 \text{ is the set defining the equational theory and } s = t \text{ is the conjecture.}$$

$$\textbf{Orient}_L\ \frac{\{u = v\} \cup E, R, C}{E, \{u \overset{\pm}{\to} v\} \cup R, C}\ \text{ if } u^* > v^* \qquad \textbf{Orient}_R\ \frac{\{u = v\} \cup E, R, C}{E, \{v \overset{-}{\to} u\} \cup R, C}\ \text{ if } v^* > u^*$$

$$\textbf{Orient'}\ \frac{\{u = v\} \cup E, R, C}{E, \{u \overset{\pm}{\to} v; v \overset{-}{\to} u\} \cup R, C}\ \text{ if } u^* \text{ and } v^* \text{ are not comparable w.r.t. to } >.$$

$$\textbf{Rewrite}_L\ \frac{\{u = v\} \cup E, \{l \overset{\pm}{\to} r\} \cup R, C}{\{Rew(u, \xrightarrow[p,\sigma]{l \overset{\pm}{\to} r}) = v\} \cup E, \{l \overset{\pm}{\to} r\} \cup R, C}\ \text{ if } u^*|_p = l^*\sigma \text{ and } l^*\sigma > r^*\sigma.$$

$$\textbf{Rewrite'}_L\ \frac{E, \{l \overset{\pm}{\to} r\} \cup R, \{s = t\} \cup C}{E, \{l \overset{\pm}{\to} r\} \cup R, \{Rew(s, \xrightarrow[p,\sigma]{l \overset{\pm}{\to} r}) = t\} \cup C}\ \text{ if } s^*|_p = l^*\sigma \text{ and } l^*\sigma > r^*\sigma.$$

Rewrite$_R$ **Rewrite'**$_R$

$$\textbf{Collapse}_+\ \frac{E, \{l \overset{\pm}{\to} r; g \overset{\pm}{\to} d\} \cup R, C}{\{Rew(l, \xrightarrow[p,\sigma]{g \overset{\pm}{\to} d}) = r\} \cup E, \{g \overset{\pm}{\to} d\} \cup R, C}\ \text{ if } l^*|_p = g^*\sigma.$$

$$\textbf{Collapse}_-\ \frac{E, \{l \overset{-}{\to} r; g \overset{\pm}{\to} d\} \cup R, C}{\{r = Rew(l, \xrightarrow[p,\sigma]{g \overset{\pm}{\to} d})\} \cup E, \{g \overset{\pm}{\to} d\} \cup R, C}\ \text{ if } l^*|_p = g^*\sigma.$$

$$\textbf{Compose}\ \frac{E, \{l \overset{\pm}{\to} r; g \overset{\pm}{\to} d\} \cup R, C}{E, \{l \overset{\pm}{\to} Rew(r, \xrightarrow[p,\sigma]{g \overset{\pm}{\to} d}); g \overset{\pm}{\to} d\} \cup R, C}\ \text{ if } r^*|_p = g^*\sigma.$$

$$\textbf{Critical pair}\ \frac{E, \{l \overset{\pm}{\to} r; g \overset{\pm}{\to} d\} \cup R, C}{\left\{\begin{array}{c} l^*\rho_1[d^*\rho_2]_p\sigma = r^*\rho_1\sigma \\ \text{by } \kappa(\xleftarrow[p]{g \overset{\pm}{\to} d} l\rho_1\sigma \xrightarrow[\Lambda]{l \overset{\pm}{\to} r}) \end{array}\right\} \cup E, \{l \overset{\pm}{\to} r; g \overset{\pm}{\to} d\} \cup R, C}\ \text{ if } l^*\rho_1|_p\sigma = g^*\rho_2\sigma.$$

$$\textbf{Narrow}_L\ \frac{E, \{g \overset{\pm}{\to} d\} \cup R, \{s = t\} \cup C}{E, \{g \overset{\pm}{\to} d\} \cup R, \left\{\begin{array}{c} s^*\rho_1[d^*\rho_2]_p\sigma = t^*\rho_1\sigma \\ \text{by } \nu_L(\xleftarrow[p]{g \overset{\pm}{\to} d} s\rho_1\sigma \xrightarrow[\Lambda]{s \overset{\pm}{\to} t}) \end{array}\right\} \cup \{s = t\} \cup C}\ \text{ if } s^*\rho_1|_p\sigma = g^*\rho_2\sigma.$$

$$\textbf{Narrow}_R\ \frac{E, \{g \overset{\pm}{\to} d\} \cup R, \{s = t\} \cup C}{E, \{g \overset{\pm}{\to} d\} \cup R, \left\{\begin{array}{c} s^*\rho_1\sigma = t^*\rho_1[d^*\rho_2]_p\sigma \\ \text{by } \nu_R(\xleftarrow[p]{g \overset{\pm}{\to} d} t\rho_1\sigma \xrightarrow[\Lambda]{t \overset{-}{\to} s}) \end{array}\right\} \cup \{s = t\} \cup C}\ \text{ if } t^*\rho_1|_p\sigma = g^*\rho_2\sigma.$$

$$\textbf{Success}\ \frac{E, R, \{s = t\} \cup C}{True}\ \text{ if } s^* \text{ and } t^* \text{ are unifiable.}$$

Fig. 4. Annotated rules for ordered completion.

Now each critical pair can be seen as a lemma stating a universally quantified equality between two terms, and can be proven by using either the original equalities in E_0 or the previous lemmata.

When the trace of $l \xrightarrow{\pm} r$ is $\omega(u = v)$, this means that the rule is obtained from the original equation $u = v$ of E_0 by possibly rewriting both left and right hand sides. l^* and r^* can be proven equal by the sequence of equational steps obtained by the concatenation of **1.** the reverted history of l, **2.** $u = v$ at the top with the identity substitution either forward when the rule has the same orientation as it parent equation ($+$ case) or backward when the orientation is different ($-$ case), **3.** the history of r.

When the trace of a rule $l \xrightarrow{\pm} r$ is $\kappa(\xleftarrow[p]{l_2 \xrightarrow{\pm} r_2} l_1\sigma \xrightarrow[\Lambda]{l_1 \xrightarrow{\pm} r_1})$, the rule is obtained from a critical pair between two rules and possibly a change of orientation. l^* and r^* can be proven equal by the sequence of equational steps obtained from the concatenation of **1.** the reverted history of l, **2.** a proof between l^0 and r^0 depending on the orientation; when there is no change of orientation, **2.a** $l_2 \xrightarrow{\pm} r_2$ applied backward at position p with the substitution σ_2 such that $l_2^*\sigma_2 = l_1^*\sigma|_p$ and $r_2^*\sigma_2 = l^0|_p$, **2.b** $l_1 \xrightarrow{\pm} r_1$ applied forward at the top with the substitution σ_1 such that $l_1^*\sigma_1 = l_1^*\sigma$ and $r_1^*\sigma_1 = r^0$, (when there is a change of orientation, **2.a'** $l_1 \xrightarrow{\pm} r_1$ applied backward at the top with the substitution σ_1 such that $l_1^*\sigma_1 = l_1^*\sigma$ and $r_1^*\sigma_1 = l^0$, **2.b'** $l_2 \xrightarrow{\pm} r_2$ applied forward at position p with the substitution σ_2 such that $l_2^*\sigma_2 = l_1^*\sigma|_p$ and $r_2^*\sigma_2 = r^0|_p$), **3.** the history of r.

The proof of the original conjecture $s^0 = t^0$ is then given by the concatenation of the history of s and the reverted history of t.

Unfolding the Proof The main proof between s^0 and t^0 can be unfolded: as soon as a rule is used between u and v, at position p with the substitution σ, this rule is recursively replaced by its proof, plugged into the context $u[\_]_p$ when used forward or the context $v[\_]_p$ when used backward, and the whole sequence being instantiated by σ.

Unifiability Problem In this case the narrowing rules may apply and the proof can be given as a sequence of lemmata, or as a substitution and a sequence of equational steps using only E_0.

There are two kinds of lemmata, those coming from critical pairs as above, and those coming from narrowing. Again, the useful ones can be extracted from the conjecture $s = t$ which triggered the **Success** rule.

Critical Pairs and Narrowing Steps as Cuts The computation of the set of useful lemmata is similar as in the word case when the trace of $s = t$ is of the form $\omega(\_)$, but the starting set has to be extended with $l_1 \xrightarrow{\pm} r_1$ and $l_2 \xrightarrow{\pm} r_2$ when the trace is of the form $\nu_{L/R}(\xleftarrow[p]{l_2 \xrightarrow{\pm} r_2} l_1\sigma \xrightarrow[\Lambda]{l_1 \xrightarrow{\pm} r_1})$. This set can be sorted as above, and the proof of a critical pair lemma is exactly the same. The proof of a narrowing lemma is different: let $s' = t'$ be a conjecture obtained by narrowing from the rule $g \xrightarrow{\pm} d$ and the conjecture $s = t$. From the formal proof point of view, this means that the goal $s = t$ has been replaced by $s' = t'$; one has to demonstrate that the replacement is sound, that is $s' = t'$ implies $s = t$. Hence the proof of $s = t$ is actually a sequence of rewriting steps between s^*

and t^* instantiated by the appropriate substitution, using $s' = t'$ and some smaller critical pair lemmata. In the case of left narrowing, for example, the substitution is equal to σ_1 such that $s^*\sigma_1 = s^*\sigma$ and $t^*\sigma_1 = t'^0$ and the sequence of rewriting steps is obtained by the concatenation of **1.** the forward application of the rule $g \xrightarrow{\pm} d$ at position p with the substitution σ_2 such that $g^*\sigma_2 = s^*\sigma|_p$ and $d^*\sigma_2 = s'^0|_p$, **2.** the history of s', **3.** the forward application of $s' = t'$ at the top with the identity substitution, **4.** the reverted history of t'. The case of right narrowing is similar, and the case of the original conjecture has already been described in the word problem case.

The proof of the last goal, that is the conjecture $s = t$ which actually triggered the **Success** rule is the substitution σ which unifies s^* and t^*, and the sequence of rewriting steps obtained by the concatenation of the history of s and the reverted history of t, every step being then instantiated by σ.

Unfolding the Proof As in the word problem case, the main proof can also be unfolded, but one has also to propagate the substitution introduced by each narrowing lemma.

4 Translation of CiME Proofs into Reified Proofs

First we describe the way we model unifiability problems inside Coq, then in Subsection 4.2 we explain how to reify a single sequence of rewrite steps in an suitable context, and finally in Subsections 4.3 and 4.4 how to obtain a global proof from several lemmata by building the needed contexts, and by combining the small proofs together.

4.1 Modelling Unifiability Problems inside Coq

A natural way to represent unifiability problems inside Coq is to represent algebraic terms with Coq terms (deep embedding), but since Coq has a typed language we first need to suppose we have a type `domain` in the Coq sort Set.

This encoding is not harmless at all since Coq types are not inhabited by default, which contradicts the semantics of unifiability problems: with an empty domain, the provability of the conjecture $f(x) = f(x)$ (which is implicitly existentially quantified) depends on the existence of a constant symbol in the signature. Therefore we assume we have a `dummy` constant in our `domain` type to model the non-emptiness assumption.

We need to introduce a Coq object of type $\overbrace{\texttt{domain} \to \cdots \to \texttt{domain}}^{n \text{ times}} \to \texttt{domain}$ for every n-ary function symbol in our signature. To represent the equality predicate we use Coq's standard polymorphic equality which is the interpretation of the $\doteq$ construction. Since we always use the `domain` type, with index 1 in our domain signature, 1 will be the default subscript for $\doteq$, $\dot\forall$ and $\dot\exists$. For each equality $s = t$ we choose an arbitrary total ordering on variables. We suppose rules will always be quantified in increasing order with respect to this ordering.

A unifiability problem is formed by a list of universally quantified equalities $R_1, \ldots, R_n$ which are declared as Coq hypotheses and an existentially quantified equality G which is declared as the goal we want to prove.

4.2 Sequence of Rewrite Steps

In a tool like CiME, the equations and the rewrite rules are implicitly universally quantified and some new variables are created when needed, whereas in the intermediate sequent calculus, any variable has to come from a context. The main difficulty of the subsection is to fill this gap by explaining in which contexts we are able to reify a sequence of rewrite steps.

Let $R_1, \ldots, R_n$ be rules, suppose CiME has given a rewriting trace between s_1 and t as follows: $s_1 \xrightarrow[\sigma_1,p_1]{\pm R_1} s_2 \xrightarrow[\sigma_2,p_2]{\pm R_2} \cdots \xrightarrow[\sigma_n,p_n]{\pm R_n} s_{n+1} = t$. We say Γ is an *adapted context* for this trace if it contains a representation of $y_1, \ldots, y_m$, the free variables in s_1 and t, a representation of $\texttt{dummy}$, and the closed hypotheses $\dot{R}_1, \ldots, \dot{R}_n$. In such a context Γ, the reification of an open term t is defined by if $t = (f\ a_1 \ldots a_p)$ then $\dot{t} = (\dot{f}\ \dot{a}_1 \ldots \dot{a}_p)$, otherwise if $t = y_i$ then $\dot{t}$ is the corresponding variable in Γ, and if t is an unknown variable then $\dot{t} = \dot{\texttt{dummy}}$. These unknown variables appear for example when the conjecture $f(a,a) = f(b,b)$ is proven using the hypothesis $\forall xy.f(x,x) = y$.

Theorem 2. *There exists a proof-trace of $\Gamma \vdash \dot{s}_1 \doteq \dot{t}$ for any adapted context Γ.*

Proof. We prove by downwards induction on i that for any adapted context Γ there exist π such that $\pi : \Gamma \vdash \dot{s}_i \doteq \dot{t}$:

If $i = n+1$ then s_i is t, so we have $=_I\ :\Gamma \vdash (t \doteq t)$ for any Γ.

Otherwise, let Γ be an adapted context of length h and k be the index of $\dot{R}_i$ in Γ, let $z_1, \ldots, z_l$ be the free variables in R_i. We can build the following proof tree:

$$\dfrac{\begin{array}{c}\text{Induction hypothesis with } \Gamma; h{+}1 : R_i\{z_1 \dot{\mapsto} z_1\sigma_i\} \ldots ; h{+}l : \dot{R_i\sigma_i} \\ \vdots \\ \pi : \Gamma; \ldots ; h{+}l : \dot{R_i\sigma_i} \vdash \dot{s_{i+1}} \doteq \dot{t}\end{array}}{\begin{array}{c}(=_{E_1} (h{+}l) \leftrightarrow \dot{p}_i\ \pi) : \Gamma; \ldots ; h{+}l : \dot{R_i\sigma_i} \vdash \dot{s}_i \doteq \dot{t} \\ \vdots\ l\ \forall_E \text{ steps} \\ (\forall_E\ k\ \dot{z_1\sigma_i}(\forall_E\ (h{+}1)\ \dot{z_2\sigma_i} \ldots (\forall_E\ (h{+}l{-}1)\ \dot{z_l\sigma_i}(=_{E_1} (h{+}l) \leftrightarrow p_i\ \pi)) \ldots) : \Gamma \vdash \dot{s}_i \doteq \dot{t}\end{array}}$$

The induction step is valid since any extension of an adapted context stays adapted. □

4.3 Closed Goals and Critical Pairs

Theorem 3. *Let G be a closed goal, $O_1, \ldots, O_n$ some original rules and $C_1, \ldots, C_m$ an ordered list of critical pairs used in the CiME trace of G. Let Γ^k be the context $\Box; \dot{\texttt{dummy}}; \dot{O}_1; \ldots ; \dot{O}_n; \dot{C}_1; \ldots ; \dot{C}_k$ with $0 \leq k \leq m$. There is a proof trace π for $\Gamma^0 \vdash \dot{G}$.*

Proof. We build inductively a proof trace π for the sequent $\Gamma^i \vdash \dot{G}$, starting from $i = m$ and going backwards to $i = 0$.

- We can build a proof trace π such that $\pi : \Gamma^m \vdash \dot{G}$ using the Theorem 2 and the trace given by CiME for G.
- Suppose we have a proof trace π such that $\pi : \Gamma^i \vdash \dot{G}$, and suppose C_i is of the form $\forall x_1 \ldots x_p.s = t$, we build the following tree to obtain a proof of $\Gamma^{i-1} \vdash G$:

$$\dfrac{\begin{array}{c}\text{Theorem 2} \\ \vdots \\ \pi' : \Gamma^{i-1}; \dot{x}_1; \ldots ; \dot{x}_p \vdash \dot{s} \doteq \dot{t} \\ \vdots\ p\ \forall_I \text{ steps} \\ (\forall_I\ \ldots (\forall_I\ \pi') \ldots) : \Gamma \vdash \dot{\forall}_1 \ldots \dot{\forall}_1 \dot{s} \doteq \dot{t}\end{array} \qquad \begin{array}{c}\text{Induction hypothesis} \\ \vdots \\ \pi : \Gamma^i \vdash \dot{G}\end{array}}{(\text{Cut } \dot{C}_i\ (\forall_I\ \ldots (\forall_I\ \pi') \ldots)\ \pi) : \Gamma^{i-1} \vdash \dot{G}}$$

□

4.4 Open Goals and Narrowings

Open Goals When the goal is of the form $\exists x_1, \ldots, x_n.s = t$, CiME provides a substitution σ and a rewriting trace for $s\sigma = t\sigma$. Using Theorem 2 we build a trace π for $\dot{s\sigma} \doteq \dot{t\sigma}$. The proof trace for the quantified goal is $(\exists_I\ \dot{x_1\sigma} \ldots (\exists_I\ \dot{x_n\sigma}\ \pi) \ldots)$.

Narrowings Assume the current goal is an equality $\exists x_1, \ldots, x_n.s = t$ and CiME gives a proof trace with a narrowing $N = \exists y_1, \ldots, y_m.s' = t'$. We do a cut on $\dot{N}$. In the left premise, the goal becomes $\dot{N}$ and we build here the trace for the rest of the lemmata. In the right premise, we apply m times the $\exists_E$ rule to obtain the hypotheses $\dot{y}_1, \ldots, \dot{y}_m, \dot{s}' \doteq \dot{t}'$, and we are in the case of the open goal described above.

5 Benchmarks

We run successfully CiME and Coq together on 230 problems coming from the TPTP library [16]. These problems are a subset of the 778 unifiability and word problems of TPTP. Some of the 778 TPTP problems are discarded because they are not solved within the given time (298), some others because they do not have a positive answer (11) or because they involve AC symbols (239), not handled by our framework yet.

The experiments were made on a 1.8GHz Pentium PC with 1Gb RAM, and a timeout of 600s on the completion process. For each of the 230 completion successes, 4 proofs were automatically generated, a short reified proof, a short proof with tactics, a cut-free reified proof and a cut-free proof with tactics. We used the current CVS versions of CiME3 and Coq, with the virtual machine turned on, which helps the Coq kernel reduce terms (see [9]).

Coq has been run on each of the 4*230 generated proofs, again with a timeout of 600s. We have observed that the short proofs are always checked in less that 1 second, for reified proofs as well as for tactics proofs, whatever the completion time. A short reified proof takes less time than the corresponding short tactics proof, but this is on very short times, so not very significant. The cut-free reified proofs take less time than the cut-free tactics, and the factor varies between 1 and 30. There is even an example (GRP614-1) where the reified proof is checked in 2 seconds and Coq gives up on the tactics proof. Some of the cut-free proofs are actually huge (several millions of lines) and cannot be handled by Coq (14 reified proofs and 16 script proofs).

6 Conclusion

We have described how to use reflection for proofs in Coq, how to annotate the usual ordered completion rules in order to build a trace, and how to turn the obtained trace into a reflective proof. The experiments made so far have validated the reflective approach and shown that some automation may be introduced in Coq and release the user from a part of the proof.

Previous works on reflection either aimed at proving meta-properties of proof trees in a very general framework [11] or at actually solving domain specific problems and at providing some automation for interactive provers [3, 6]. We claim that our work belongs to the second trend but without loss of generality since our development is parameterized by the signature. Special care has been devoted to efficiency of proof-checking functions written in the Coq language.

We plan to work in several directions. First finalize the existing implementation as a Coq tactic by adding glue code, then extend the reflection mechanism to other calculi; for example $LJTI$ which adds arbitrary non-recursive connectives to first-order logic with a contraction-free presentation [5], or classical multi-succedent calculi to handle more general traces produced *e.g.* by classical tableaux provers. Finally handle AC function symbols by reflecting AC-steps.

References

1. C. Alvarado. *Réflexion pour la réécriture dans le calcul des constructions inductives*. PhD thesis, Université Paris-Sud, Dec. 2002.
2. L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures 2: Rewriting Techniques*, chapter 1, pages 1–30. Academic Press, New York, 1989.
3. M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3):253–275, 2002.
4. E. Contejean, C. Marché, and X. Urbain. CiME3, 2004. `http://cime.lri.fr/`.
5. P. Corbineau. First-order reasoning in the Calculus of Inductive Constructions. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES 2003 : Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 162–177, Torino, Italy, Apr. 2004. Springer-Verlag.
6. P. Crégut. Une procédure de décision réflexive pour l'arithmétique de Presburger en Coq. Deliverable, Projet RNRT Calife, 2001.
7. N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5):380–392, 1972.
8. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.
9. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.
10. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.
11. D. Hendriks. Proof Reflection in Coq. *Journal of Automated Reasoning*, 29(3-4):277–307, 2002.
12. T. Hillenbrand and B. Löchner. The next waldmeister loop. In A. Voronkov, editor, *Proceedings of CADE 18*, LNAI, pages 486–500. Springer-Verlag, 2002.
13. J. Hsiang and M. Rusinowitch. On word problems in equational theories. In T. Ottmann, editor, *14th International Colloquium on Automata, Languages and Programming*, volume 267 of *LNCS*, pages 54–71, Karlsruhe, Germany, July 1987. Springer-Verlag.
14. Q.-H. Nguyen. Certifying Term Rewriting Proofs in ELAN. In M. van den Brand and R. Verma, editors, *Proceedings of the International Workshop RULE'01*, volume 59 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.
15. G. E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM Journal on Computing*, 12(1):82–100, Feb. 1983.
16. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
17. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, Apr. 2004. `http://coq.inria.fr`.