Higher order proof reconstruction from paramodulation-based refutations: the unit equality case

Andrea Asperti and Enrico Tassi

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY
asperti@cs.unibo.it tassi@cs.unibo.it

Abstract. In this paper we address the problem of reconstructing a higher order, checkable proof object starting from a proof trace left by a first order automatic proof searching procedure, in a restricted equational framework. The automatic procedure is based on superposition rules for the unit equality case. Proof transformation techniques aimed to improve the readability of the final proof are discussed.

1 Introduction

The integration of technologies developed by the automatic theorem proving (ATP) community with modern interactive theorem provers seems a fruitful research objective. ATP technologies showed their effectiveness in many occasions[7, 17] and the lack of comfortable automation is one of the most commonly issues reported by users of interactive theorem provers. This challenge gets even more interesting when the target interactive theorem prover follows the independent verification principle, building proof objects that can be validated by third party checkers. Providing a valuable proof trace is not the main goal of ATP systems, and even when they do, the information can be too ambiguous to be checked by a different prover (see [5]).

Among the activities of interactive proving that one would like to be supported by powerful automation techniques a major one is rewriting. In this paper we describe our approach to this problem in relation with the interactive theorem prover Matita[1]. In particular we integrated Matita with a first order, paramodulation[11] based solver (currently restricted to the unit equality case). The solver is able to return a trace informative enough to be read back into a proof object of Matita, that is a term of the Calculus of Inductive Constructions[13, 21] (CIC). In this paper we focus on the information that must be embedded in traces, on the reconstruction of typable proof objects, and finally on the refinement of the resulting proofs to enhance readability. In particular we prove that any equational proof based on rewriting can be transformed into a transitivity chain, where each step is justified by a simple side argument (an axiom, or an already proved lemma). This format is really close to the standard mathematical display of this kind of proofs.

The paper starts introducing the interactive theorem prover Matita and giving an overview of the automatic procedure we implemented (Sec. 2). In particular Sec. 2.1 describes how the notion of equality is encoded in CIC, while Sec. 2.2 and 2.3 describe the variant of the paramodulation calculus implemented, the proof searching algorithm and the lightweight representation of proofs adopted during proof search. A proof reconstruction procedure is then presented in Sec. 3 and its result is refined with some transformations that are detailed in Sec. 4.

We will introduce notational conventions when needed, but as a general rule we will use a different syntax for functions living in the proof language CIC or living in the meta level and manipulating CIC terms. Proofs will essentially be applicative lambda terms written using the notation (f $a\ b\ c$), while we will write $\theta(a,\ b,\ c)$ for functions at the meta level.

2 Automatic proof search procedure implementation

Matita is an interactive theorem prover under development at the university of Bologna (see [1] for a description of the innovative features of the system).

Matita is based on the Curry-Howard isomorphism, adopting the Calculus of Inductive Constructions as its logical framework.

The automatic proof search procedure is a component of Matita, but is essentially orthogonal to the rest of the system. It has been extensively tested with unit equality problems of the TPTP[18] library. The results obtained by the procedure can be browsed on TPTP website[1] (we solve 512 problems out of 700 in the standard TPTP time limit of 10 minutes).

CIC terms are translated into first order terms by a forgetful procedure that simply erases all type information, and transforms into opaque constants all terms not belonging to the first order framework (fixpoints, pattern matching terms, etc.).

The inverse transformation takes advantage by the so called *refiner*, that is a type inference procedure typical of higher order interactive provers.

An overview of the rules used by the solver is given in Section 2.2. These rules are decorated with proofs; the next section gives the few notions needed to understand the proof terms.

2.1 Rewriting in the calculus of inductive constructions

In the calculus of inductive constructions, equality is not a primitive notion, but it is defined as the smallest predicate containing (induced by) the reflexivity principle.

$$\text{Inductive eq } (A : \mathit{Type})\ (x : A) : A \to \mathit{Prop} \overset{\text{def}}{=} \text{refl\_eq} : \text{eq } A\ x\ x.$$

For the sake of readability we will use the notation $a_1 =_A a_2$ for (eq $A\ a_1\ a_2$).

[1] http://www.cs.miami.edu/∼tptp/

As a consequence of this inductive definition, and similarly to all inductive types, it comes equipped with an elimination principle named eq_ind that, for any type A, any elements a_1, a_2 of A, any property P over A, given a proof h of $(P\ a_1)$ and a proof k that $a_1 =_A a_2$ gives back a proof of $(P\ a_2)$.

$$\frac{h : P\ a_1 \qquad k : a_1 =_A a_2}{(\text{eq\_ind}\ A\ a_1\ P\ h\ a_2\ k) : P\ a_2}$$

Similarly, we may define a higher order elimination principle eq_ind_r such that

$$\frac{h : P\ a_2 \qquad k : a_1 =_A a_2}{(\text{eq\_ind\_r}\ A\ a_2\ P\ h\ a_1\ k) : P\ a_1}$$

These are the building blocks of the proofs we will generate. With this definition of equality standard properties like reflexivity, symmetry and transitivity can be easily proved and are part of the standard library of lemmas available in Matita.

2.2 Superposition rules

Paramodulation is precisely the management of equality by means of rewriting: given a formula (clause) $P(s)$, and an equality $s = t$, we may conclude $P(t)$. What makes paramodulation a really effective tool is the possibility of suitably constraining rewriting in order to avoid redundant inferences without loosing completeness. This is done by requiring that rewriting always replace *big* terms by *smaller* ones, with respect to a special ordering relation $\succ$ among terms, that satisfies certain properties, called the *reduction ordering*. This restriction of the paramodulation rule is called *superposition*.

Equations are traditionally split in two groups: facts (positive literals) and goals (negative literals). We have two basic rules: superposition right and superposition left. Superposition right combines facts to generate new facts: it corresponds to a forward reasoning step. Superposition left combines a fact and a goal, generating a new goal: logically, it is a backward reasoning step, reducing a goal G to a new one G'. The fragment of proof that can be associated to this new goal G' is thus not a proof of G' , but a proof of G *depending* on proof of G' (i.e. a proof of $G' \vdash G$).

We shall use the following notation: an equational fact will have the shape $\vdash M : e$, meaning that M is a proof of e; an equational goal will have the shape $\alpha : e \vdash M : C$, meaning that in the proof M of C the goal e is still open, i.e. M may depend on α.

Given a term t we write $t|_p$ to denote the subterm of t at position p, and $t[r]_p$ for the term obtained from t replacing the subterm $t|_p$ with r. Given a substitution σ we write $t\sigma$ for the application of the substitution to the term, with the usual meaning.

The logical rules, decorated with proofs, are the following:

Superposition left

$$\frac{\vdash h : l =_A r \qquad \alpha : t =_B s \vdash M : C}{\beta : t[r]_p\sigma =_B s\sigma \vdash M\sigma[R/\alpha\sigma] : C\sigma}$$

if $\sigma = mgu(l, t|_p)$, $t|_p$ is not a variable, $l\sigma \succ r\sigma$ and $t\sigma \succ s\sigma$; and
$R = (\text{eq\_ind\_r}\ A\ r\sigma\ (\lambda x : A.t[x]_p =_B s)\sigma\ \beta\ l\sigma\ h\sigma) : t\sigma =_B s\sigma$

Superposition right

$$\frac{\vdash h : l =_A r \qquad \vdash k : t =_B s}{\vdash R : t[r]_p\sigma =_B s\sigma}$$

if $\sigma = mgu(l, t|_p)$, $t|_p$ is not a variable, $l\sigma \succ r\sigma$ and $t\sigma \succ s\sigma$; and
$R = (\text{eq\_ind}\ A\ l\sigma\ (\lambda x : A.t[x]_p =_B s)\sigma\ k\sigma\ r\sigma\ h\sigma) : t[r]_p\sigma =_B s\sigma$

Equality resolution

$$\frac{\alpha : t =_A s \vdash M : C}{\vdash M[\text{refl\_eq}\ A\ t\sigma/\alpha] : C}$$

if there exists $\sigma = mgu(t, s)$; (notice $refl\_eq\ A\ t : t =_A t$, being refl_eq the constructor of the equality).

The main theorem is that, given a set of facts S, and a goal e, an instance e' of e is a logical consequence of S if and only if, starting from the trivial axiom $\alpha : e \vdash \alpha : e$ we may prove $\vdash M : e'$ (and in this case M is a correct proof term).

Simplification rules such as tautology elimination, subsumption and especially demodulation can be added to the systems, but they do not introduce major conceptual problems, and hence they will not be considered here.

2.3 Proof search and its representation

Given the three superposition rules above, proof search is performed using the "given clause" algorithm (see [14, 15]). The algorithm keeps all known facts and goals split in two sets: active, and passive. At each iteration, the algorithm carefully chooses an equation (given clause) from the passive set; if it is a goal (and not an identity), then it is combined via superposition left with all active facts; if it is a fact, superposition right is used instead. The selected equation is added to the (suitable) active set, while all newly generated equations are added to the passive set, and the cycle is repeated.

As the reader may imagine a huge number of equations is generated during the proof search process, but only few of them will be actually used to prove the goal. Even if demodulation and subsumption are effective tools to discard equations without loosing completeness, all automatic theorem provers adopt clever techniques to strike down the space consumption of each equation. This usually leads to an extensive use of sharing in the data structures, and to drop the idea of carrying a complete proof representation in favor of recording a minimal and lightweight proof trace. The latter choice is usually not a big concern for ATP systems, since proofs are mainly used for debugging purposes, but for an interactive theorem prover that follows the independent verification principle like Matita, proof objects are essential and thus it must be possible to reconstruct a complete proof object in CIC from the proof trace.

In our implementation the proof trace is composed by two slightly different kind of objects, corresponding to the two superposition steps. Superposition right steps are encoded with the following tuple:

$$\text{type } rstep \stackrel{\text{def}}{=} ident * ident * direction * substitution * predicate$$

The two identifiers are unambiguous names for the equations involved (h and k in the former presentation of the superposition rule), *direction* can be either Left or Right, depending if h has been used left to right or right to left (i.e. if a symmetry step has to be kept into account). The *substitution* and the *predicate* are respectively the σ (i.e. the most general unifier between l and $t|_p$) and the predicate used to build the proof R (i.e. the third element applied to eq_ind), that is essentially a representation of the position $|_p$ identifying the subterm of t that has been rewritten with r once l and $t|_p$ were unified via σ.

This representation of the predicate is not optimal in terms of space consumption; we have chosen this representation mainly for simplicity, and left the implementation of a more compact coding as a future optimization.

The representation of a superposition left step is essentially the same, but the second equation identifier has been removed, since it implicitly refers to the goal. We will call the type of these steps *lstep*.

A map $\Sigma : ident \rightarrow (pos\_literal * rstep)$ from identifiers to pairs of positive literal (i.e. something of the form $\vdash a =_A b$) and proof step represents all the forward reasoning performed during proof search, while a list Λ of *lstep* together with the initial goal (a negative literal) represent all backward reasoning steps.

3 Proof reconstruction

The functions defined in Fig. 1 build a CIC proof term given the initial goal g, Σ and Λ. We use the syntax "let $(\vdash l =_A r,\ \pi_h) = \Sigma(h)$ in" for the irrefutable pattern matching construct "match $\Sigma(h)$ with $(\vdash \text{eq } A\ l\ r),\ \pi_h \Rightarrow$".

The function ϕ produces proofs corresponding to application of the superposition right rule, with the exception that if h is used right to left and eq_ind_r is used to represent the hidden symmetry step. ψ builds proofs associated with the application of the superposition left rule, and fires ϕ to build the proof of the positive literal h involved.

Unfortunately this simple structurally recursive approach has the terrible behavior of inlining the proofs of positive literals even if they are used non linearly. This may (and in practice does) trigger an exponential factor in the size of proof objects. The obtained proof object is thus of a poor value, because type checking it would require an unacceptable amount of time.

As an empirical demonstration of that fact we report in Fig. 2 a graphical representation of the proof of problem GRP001-4 available in the TPTP[18] library version 3.1.1. Axioms are represented in squares, while positive literals have a circular shape. The goal is an hexagon.

Every positive literal points to the two used as hypothesis in the corresponding application of the superposition right rule. In this example a, b, c and e are

```
$\phi(\Sigma,\ (h,\ k,\ dir,\ \sigma,\ P)) =$
    let $(\vdash l =_A r,\ \pi_h) = \Sigma(h)$ and $(\vdash t =_B s,\ \pi_k) = \Sigma(k)$ in
    match $dir$ with
    | Left $\Rightarrow$ eq_ind $A\ l\sigma\ P\sigma\ \phi(\Sigma,\ \pi_k)\sigma\ r\sigma\ \phi(\Sigma,\ \pi_h)\sigma$
    | Right $\Rightarrow$ eq_ind_r $A\ r\sigma\ P\sigma\ \phi(\Sigma,\ \pi_k)\sigma\ l\sigma\ \phi(\Sigma,\ \pi_h)\sigma$

$\psi'(\Sigma,\ (h,\ dir,\ \sigma,\ P),\ (t =_B s,\ \pi_g)) =$
    let $(\vdash l =_A r,\ \pi_h) = \Sigma(h)$ in
    match $dir$ with
    | Left $\Rightarrow (P\ r)\sigma,$ eq_ind $A\ l\sigma\ P\sigma\ \pi_g\sigma\ r\sigma\ \phi(\Sigma,\ \pi_h)\sigma$
    | Right $\Rightarrow (P\ l)\sigma,$ eq_ind_r $A\ r\sigma\ P\sigma\ \pi_g\sigma\ l\sigma\ \phi(\Sigma,\ \pi_h)\sigma$

$\psi(g,\ \Lambda,\ \Sigma) =$
    let $(t =_B s) \vdash \_ = g$ in
    $snd$(fold_right$(\lambda x.\lambda y.\psi'(\Sigma,\ x,\ y),\ (t =_B s,$ refl_eq $A\ s),\ \Lambda))$

$\tau \overset{\text{def}}{=} term$
$\phi : (ident \to (pos\_literal * rstep)) * rstep \to \tau$
$\psi' : (ident \to (pos\_literal * rstep)) * lstep * (\tau * \tau) \to \tau$
$\psi : neg\_literal * lstep$ list $* (ident \to (pos\_literal * rstep)) \to \tau$
fold_right : $(lstep * (\tau * \tau) \to (\tau * \tau)) * (\tau * \tau) * lstep$ list $\to (\tau * \tau)$
```

Fig. 1. Proof reconstruction

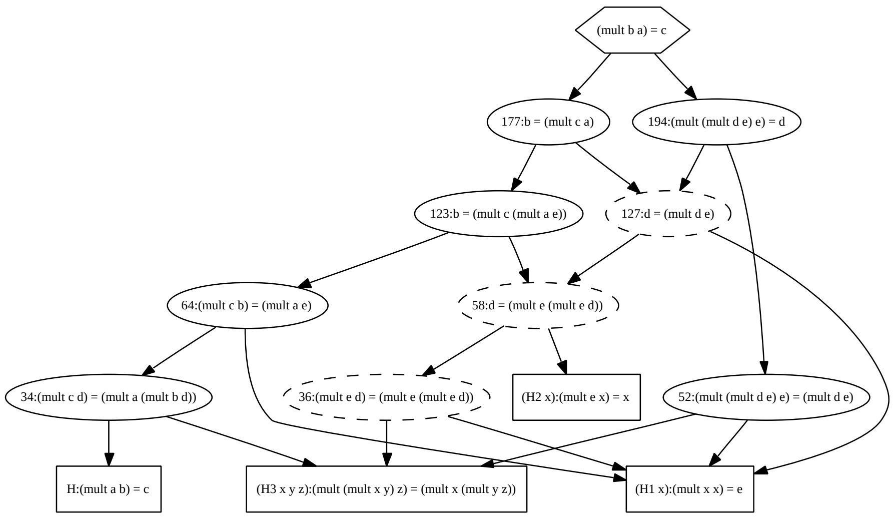

(mult b a) = c
177:b = (mult c a)
194:(mult (mult d e) e) = d
123:b = (mult c (mult a e))
127:d = (mult d e)
64:(mult c b) = (mult a e)
58:d = (mult e (mult e d))
34:(mult c d) = (mult a (mult b d))
36:(mult e d) = (mult e (mult e d))
(H2 x):(mult e x) = x
52:(mult (mult d e) e) = (mult d e)
H:(mult a b) = c
(H3 x y z):(mult (mult x y) z) = (mult x (mult y z))
(H1 x):(mult x x) = e

Fig. 2. Proof representation (shared nodes)

constants, the latter has the identity properties (axiom H2). The thesis is that a group (axioms H3, H2) in which the square of each element is equal to the unit (axiom H1) is abelian (compose H with the goal to obtain the standard formulation of the abelian predicate). Equation 127 is used twice, 58 is used three times (two times by 127 and one by 123), consequently also 36 is not used linearly. In this scenario, the simple proof reconstruction algorithm inflates the proof term, replicating the literals marked with a dashed line.

The benchmarks reported in Tab. 1show that this exponential behavior makes proof objects practically untractable. The first column reports the time the automatic procedure spent in searching the proof, and the second one the number of iterations of the given clause algorithm needed to find a proof. The amount of time necessary to typecheck a non optimized proof is dramatically bigger then the time that is needed to find the proof. With the optimization we describe in the following paragraph typechecking is as fast as proof search for easy problems like the ones shown in Tab. 1.As one would expect, when problems are more challenging, the time needed for typechecking the proof is negligible compared to the time needed to find the proof.

Problem	Search	Steps	Typing		Proof size	
			raw	opt	raw	opt
BOO069-1	2.15	27	79.50	0.23	3.1M	29K
BOO071-1	2.23	27	203.03	0.22	5.4M	28K
GRP118-1	0.11	17	7.66	0.13	546K	21K
GRP485-1	0.17	47	323.35	0.23	5.1M	33K
LAT008-1	0.48	40	22.56	0.12	933K	19K
LCL115-2	0.81	52	24.42	0.29	1.1M	37K

Tab. 1. Timing (in seconds) and proof size

Fortunately CIC provides a construct for local definitions LetIn : $ident * term * term \to term$ that is type checked efficiently: the type of the body of the definition is computed once and then stored in the context used to type check the rest of the term.

We can thus write a function that, counting the number of occurrences of each equation, identifies the proofs that have to be factorised out. In Fig. 3 the function γ returns a map from identifiers to integers. If this integer is greater than 1, then the corresponding equation will be factorised. In the example above, 127 and 58 should be factorised, since γ evaluates to two on them, and they must be factorised in this precise order, so that the proof of 127 can use the local definition of 58. The right order is the topological one, induced by the dependency relation shown in the graph.

Every occurrence of an equation may be used with a different substitution, that can instantiate free variables with different terms. Thus it is necessary to factorise closed proofs obtained λ-abstracting their free variables, and applying them to the same free variables where they occur before applying the local substitution. For example, given a proof π whose free variables are $x_1 \dots x_n$ respectively of type $T_1 \dots T_n$ we generate the following let in:

$$\text{LetIn } h \overset{\text{def}}{=} (\lambda x_1 : T_1, \dots \lambda x_n : T_n, \pi) \text{ in}$$

and the occurrences of π will look like $(h\ x_1\ \dots\ x_n)\sigma$ where σ will eventually differ.

```
$\delta'(\Sigma,\ h,\ f) =$
    let $g = (\lambda x.\text{if } x = h \text{ then } 1 + f(x) \text{ else } f(x))$ in
    if $f(h) = 0$ then
        let $(\_,\ \pi_h) = \Sigma(h)$ in
        let $(k_1,\ k_2,\ \_,\ \_,\ \_) = \pi_h$ in
        $\delta'(\Sigma,\ k_1,\ \delta'(\Sigma,\ k_2,\ g))$
    else $g$

$\delta(\Sigma,\ (h,\ \_,\ \_,\ \_),\ f) = \delta'(\Sigma,\ h,\ f)$

$\gamma(\Lambda,\ \Sigma) = \text{fold\_right}(\lambda x.\lambda y.\delta(\Sigma,\ x,\ y),\ \lambda x.0,\ \Lambda)$

$\delta' : (\mathit{ident} \to (\mathit{pos\_literal} * \mathit{rstep})) * \mathit{ident} * (\mathit{ident} \to \mathit{int}) \to (\mathit{ident} \to \mathit{int})$
$\delta : (\mathit{ident} \to (\mathit{pos\_literal} * \mathit{rstep})) * \mathit{lstep} * (\mathit{ident} \to \mathit{int}) \to (\mathit{ident} \to \mathit{int})$
$\gamma : \mathit{lstep}\ \text{list} * (\mathit{ident} \to (\mathit{pos\_literal} * \mathit{rstep})) \to (\mathit{ident} \to \mathit{int})$
```

Fig. 3. Occurrence counting

3.1 Digression on dependent types

ATP systems usually operate in a first order setting, where all variables have the same type. CIC provides dependent types, meaning that in the previous example the type T_n can potentially depend on the variables $x_1 \ldots x_{n-1}$, thus the order in which free variables are abstracted is important and must be computed keeping dependencies into account.

Consider the case, really common in formalisations of algebraic structures, where a type, functions over that type and properties of these operations are packed together in a structure. For example, defining a group, one will probably end up having the following constants:

$$\begin{array}{rl} \text{carr} : \mathit{Group} \to \mathit{Type} & \text{inv} : \forall g : \mathit{Group}, \text{carr } g \to \text{carr } g \\ \text{e} : \forall g : \mathit{Group}, \text{carr } g & \text{mul} : \forall g : \mathit{Group}, \text{carr } g \to \text{carr } g \to \text{carr } g \\ & \text{id\_l} : \forall g : \mathit{Group}, \forall x : \text{carr } g, \text{mul } g\ (\text{e } g)\ x = x \end{array}$$

Saturation rules work with non abstracted (binder free) equations, thus the id_l axiom is treated as $(\mathit{mul}\ x\ (e\ x)\ y = y)$ where x and y are free. If these free variables are blindly abstracted, an almost ill typed term can be obtained:

$$\lambda y : ?_1, \lambda x : ?_2, \text{mul } x\ (e\ x)\ y = y$$

where there is no term for $?_1$ such that $?_1 = (\text{carr } x)$ as required by the dependency in the type of mul: the second and third arguments must have type carr of the first argument. In the case above, the variable y has a type that depends on x, thus abstracting y first, makes it syntactically impossible for its type to depend on x. In other words $?_1$ misses x in its context.

When we decided to integrate automatic rewriting techniques like superposition in Matita, we were attracted by their effectiveness and not in studying a generalisation of these techniques to a much more complex framework like

CIC. The main, extremely practical, reason is that the portion of mathematical problems that can be tackled using first order techniques is non negligible and for some problems introduced by dependent types, like the one explained above, the solution is reasonably simple. Exploiting the explicit polymorphism of CIC, and the rigid structure of the proofs we build (i.e. nested application of eq_ind) it is possible to collect free variables that are used as types, inspecting the first arguments of eq_ind and eq: these variable are abstracted first. Even if this simple approach works pretty well in practice and covers the probably most frequent case of type dependency, it is not meant to scale up to the general case of dependent types, in which we are not interested.

4 Proof refinement

Proofs produced by paramodulation based techniques are very difficult to understand for a human. Although the single steps are logically trivial, the overall design of the proof is extremely difficult to grasp. This need is also perceived by the ATP community; for instance, in order to improve readability, the TPTP[18] library, provides a functionality to display proofs in a graphical form (called YuTV), pretty similar to the one in Fig. 2.

In the case of purely equational reasoning, mathematicians traditionally organize the proof as a chain of rewriting steps, each one justified by a simple side argument (an axiom, or an already proved lemma). Technically speaking, such a chain amounts to a composition of transitivity steps, where as proof leaves we only admit axioms (or their symmetric variants), possibly contextualized.
Formally, the basic components we need are provided by the following terms:

$$\begin{array}{l}\mathrm{trans} : \forall A : Type.\forall x, y, z : A.x =_A y \to y =_A z \to x =_A z \\ \mathrm{sym} : \forall A : Type.\forall x, y : A.x =_A y \to y =_A x \\ \mathrm{eq\_f} : \forall A, B : Type.\forall f : A \to B.\forall x, y : A.x =_A y \to (f\ x) =_B (f\ y)\end{array}$$

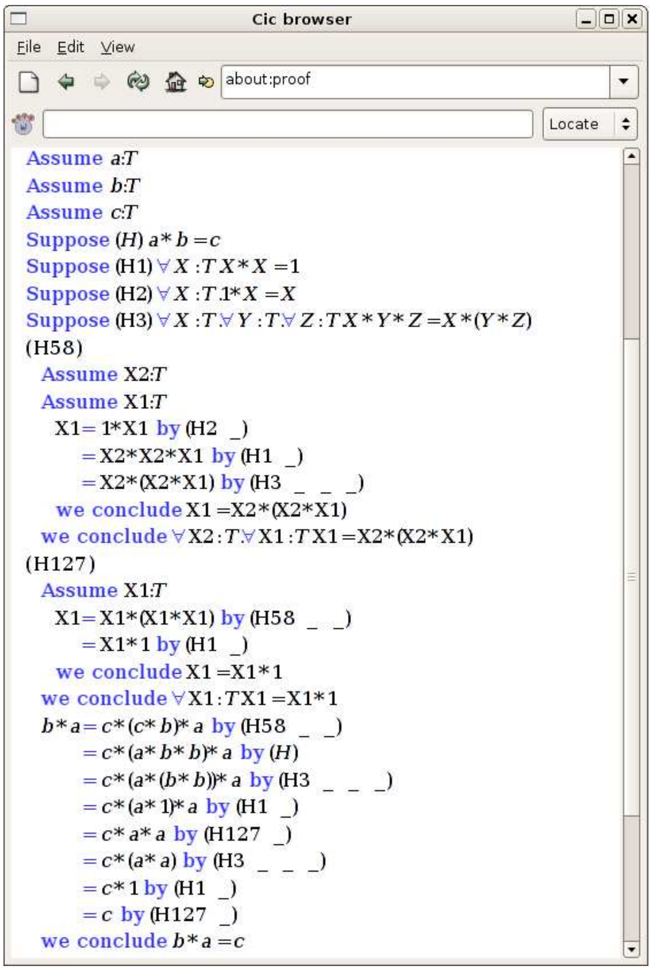
The last term (function law) allows to contextualize the equation $x =_A y$ in an arbitrary context f.
The normal form for equational proofs we are interested in is described by the following grammar:

Definition 1 (Proof normal form).

$$\begin{array}{rl}\pi = & \mathrm{eq\_f}\ B\ C\ \Delta\ a\ b\ axiom \\ | & \mathrm{eq\_f}\ B\ C\ \Delta\ a\ b\ (\mathrm{sym}\ B\ b\ a\ axiom) \\ | & \mathrm{trans}\ A\ a\ b\ c\ \pi\ \pi\end{array}$$

We now prove that any proof build by means of eq_ind and eq_ind_r may be transformed in the normal form of definition 1. The transformation is defined in two phases. In the first phase we replace all rewriting steps by means of applications of transitivity, symmetry and function law. In the second phase we propagate symmetries towards the leaves.

In Figure 4 we show an example of the kind of rendering obtained after the transformation, relative to the proof of GRP001-4.



```
Assume a:T
Assume b:T
Assume c:T
Suppose (H) a* b =c
Suppose (H1) ∀ X :T X * X =1
Suppose (H2) ∀ X :T.1* X =X
Suppose (H3) ∀ X :T.∀ Y :T.∀ Z :T X * Y * Z =X *(Y * Z)
(H58)
  Assume X2:T
  Assume X1:T
   X1= 1*X1 by (H2  _)
       =X2*X2*X1 by (H1  _)
       =X2*(X2*X1) by (H3  _  _  _)
   we conclude X1 =X2*(X2*X1)
  we conclude ∀ X2 :T.∀ X1 :T X1 =X2*(X2*X1)
(H127)
  Assume X1:T
   X1= X1*(X1*X1) by (H58  _  _)
       =X1*1 by (H1  _)
   we conclude X1 =X1*1
  we conclude ∀ X1 :T X1 =X1*1
  b* a= c* (c* b)* a by (H58  _  _)
       = c* (a* b* b)* a by (H)
       = c* (a* (b* b))* a by (H3  _  _  _)
       = c* (a* 1)* a by (H1  _)
       = c* a* a by (H127  _)
       = c* (a* a) by (H3  _  _  _)
       = c* 1 by (H1  _)
       = c by (H127  _)
  we conclude b* a =c
```

Fig. 4. Natural language rendering of the (refined) proof object of GRP001-4

4.1 Phase 1: transitivity chain

The first phase of the transformation is defined by the ρ function of Fig. 5. We use Δ and Γ for contexts (i.e. unary functions). We write $\Gamma[a]$ for the application of Γ to a, that puts a in the context Γ, and $(\Delta \circ \Gamma)$ for the composition of contexts, so we have $(\Delta \circ \Gamma)[a] = \Delta[\Gamma[a]]$. The auxiliary function ρ' takes a context $\Delta : B \to C$, a proof of $(c =_B d)$ and returns a proof of $(\Delta[c] =_C \Delta[d])$. In order to prove that ρ is type preserving, we proceed by induction on the size of the proof term, stating that if Δ is a context of type $B \to C$ and π is a term of type $a =_B b$, then $\rho'(\Delta,\ \pi) : \Delta[a] =_C \Delta[b]$.

$\rho(\pi) \rightsquigarrow \rho'(\lambda x{:}C.x,\ \pi) \qquad$ when $\pi : a =_C b$

$\rho'(\Delta,\ \text{eq\_ind}\ A\ a\ (\lambda x.\Gamma[x] =_B m)\ \pi_1\ b\ \pi_2) \rightsquigarrow$
$\quad \text{trans}\ C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \Delta[m]$
$\qquad (\text{sym}\ C\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]\ \rho'(\Delta \circ \Gamma,\ \pi_2))\ \rho'(\Delta,\ \pi_1)$

$\rho'(\Delta,\ \text{eq\_ind\_r}\ A\ a\ (\lambda x.\Gamma[x] =_B m)\ \pi_1\ b\ \pi_2) \rightsquigarrow$
$\quad \text{trans}\ C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \Delta[m]\ \rho'(\Delta \circ \Gamma,\ \pi_2)\ \rho'(\Delta,\ \pi_1)$

$\rho'(\Delta,\ \text{eq\_ind}\ A\ a\ (\lambda x.m =_B \Gamma[x])\ \pi_2\ b\ \pi_1) \rightsquigarrow$
$\quad \text{trans}\ C\ \Delta[m]\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]\ \rho'(\Delta,\ \pi_2)\ \rho'(\Delta \circ \Gamma,\ \pi_1)$

$\rho'(\Delta,\ \text{eq\_ind\_r}\ A\ a\ (\lambda x.m =_B \Gamma[x])\ \pi_1\ b\ \pi_2) \rightsquigarrow$
$\quad \text{trans}\ C\ \Delta[m]\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]$
$\qquad \rho'(\Delta,\ \pi_1)\ (\text{sym}\ C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \rho'(\Delta \circ \Gamma,\ \pi_2))$

$\rho'(\Delta,\ \pi) \rightsquigarrow \text{eq\_f}\ B\ C\ \Delta\ a\ b\ \pi \qquad$ when $\pi : a =_B b$ and $\Delta : B \to C$

Fig. 5. Transitivity chain construction

Theorem 1 (ρ' injects). *For all B and C types, for all a and b of type B, if $\Delta : B \to C$ and $\pi : a =_B b$, then $\rho'(\Delta,\ \pi) : \Delta[a] =_C \Delta[b]$*

Proof. We proceed by induction on the size of the proof term.

Base case By hypothesis we know $\Delta : B \to C$, and $\pi : a =_B b$, thus a and b have type B and ($\text{eq\_f}\ B\ C\ \Delta\ a\ b\ \pi$) is well typed, and proves $\Delta[a] =_C \Delta[b]$

Inductive case (We analyse only the first case, the others are similar) By hypothesis we know $\Delta : B \to C$, and

$$\pi = (\text{eq\_ind}\ A\ a\ (\lambda x.\Gamma[x] =_B m)\ \pi_1\ b\ \pi_2) : \Gamma[b] =_B m$$

From the type of eq_ind we can easily infer that $\pi_1 : \Gamma[a] =_B m$, $\pi_2 : a =_A b$, $\Gamma : A \to B$, $m : B$ and both a and b have type A. Since $\Delta : B \to C$, $\Delta \circ \Gamma$ is a context of type $A \to C$. Since π_2 is a subterm of π, by inductive hypothesis we have

$$\rho'(\Delta \circ \Gamma,\ \pi_2) : (\Delta \circ \Gamma)[a] =_C (\Delta \circ \Gamma)[b]$$

Since $(\Delta \circ \Gamma) : A \to C$ and a and b have type A, both $(\Delta \circ \Gamma)[a]$ and $(\Delta \circ \Gamma)[b]$ live in C. We can thus type the following application.

$$\pi_2' \overset{\text{def}}{=} (\text{sym}\ C\ (\Delta \circ \Gamma)[a]\ (\Delta \circ \Gamma)[b]\ \rho'(\Delta \circ \Gamma,\ \pi_2)) : (\Delta \circ \Gamma)[b] =_C (\Delta \circ \Gamma)[a]$$

We can apply the induction hypothesis also on $\pi_1' \overset{\text{def}}{=} (\rho'\ \Delta\ \pi_1)$ obtaining that is has type $(\Delta \circ \Gamma)[a] =_C \Delta[m]$. Since $\Delta[m] : C$, we can conclude that

$$\pi_3 \overset{\text{def}}{=} (\text{trans}\ C\ (\Delta \circ \Gamma)[b]\ (\Delta \circ \Gamma)[a]\ \Delta[m]\ \pi_2'\ \pi_1') : (\Delta \circ \Gamma)[b] =_C \Delta[m]$$

Expanding $\circ$ we obtain $\pi_3 : \Delta[\Gamma[b]] =_C \Delta[m]$

□

Corollary 1 **(ρ is type preserving).**

Proof. Trivial, since the initial context is the identity. □

4.2 Phase 2: symmetry step propagation

The second phase of the transformation is performed by the θ function in Fig.6.

$$\begin{array}{l}
\theta(\mathrm{sym}\ A\ b\ a\ (\mathrm{trans}\ A\ b\ c\ a\ \pi_1\ \pi_2)) \leadsto \\
\qquad \mathrm{trans}\ A\ a\ c\ b\ \theta(\mathrm{sym}\ A\ c\ a\ \pi_2)\ \theta(\mathrm{sym}\ A\ b\ c\ \pi_1) \\
\theta(\mathrm{sym}\ A\ b\ a\ (\mathrm{sym}\ A\ a\ b\ \pi)) \leadsto \theta(\pi) \\
\theta(\mathrm{trans}\ A\ a\ b\ b\ \pi_1\ \pi_2) \leadsto \theta(\pi_1) \\
\theta(\mathrm{trans}\ A\ a\ a\ b\ \pi_1\ \pi_2) \leadsto \theta(\pi_2) \\
\theta(\mathrm{trans}\ A\ a\ c\ b\ \pi_1\ \pi_2) \leadsto \\
\qquad \mathrm{trans}\ A\ a\ c\ b\ \theta(\pi_1)\ \theta(\pi_2) \\
\theta(\mathrm{sym}\ B\ \Delta[a]\ \Delta[b]\ (\mathrm{eq\_f}\ A\ B\ \Delta\ a\ b\ \pi)) \leadsto \\
\qquad \mathrm{eq\_f}\ A\ B\ \Delta\ b\ a\ (\mathrm{sym}\ A\ a\ b\ \pi) \\
\theta(\pi) \leadsto \pi
\end{array}$$

Fig. 6. Canonical form construction

The third and fourth case of the definition of θ are merely used to drop a redundant reflexivity step introduced by the equality resolution rule.

Theorem 2 **(θ is type preserving).** *For all A type, for all a and b of type A, if $\pi : a =_A b$, then $\theta(\pi) : a =_A b$*

Proof. We proceed by induction on the size of the proof term analysing the cases defining θ. By construction, the proof is made of nested applications of sym and trans; leaves are built with eq_f. The base case is the last one, where θ behaves as the identity and thus is type preserving. The following cases are part of the inductive step, thus we know by induction hypothesis that θ is type preserving on smaller terms.

First case By hypothesis we know that

$$(\mathrm{sym}\ A\ b\ a\ (\mathrm{trans}\ A\ b\ c\ a\ \pi_1\ \pi_2)) : a =_A b$$

thus $\pi_1 : b =_A c$ and $\pi_2 : c =_A a$. Consequently $(\mathrm{sym}\ A\ c\ a\ \pi_2) : a =_A c$ and $(\mathrm{sym}\ A\ b\ c\ \pi_1) : c =_A b$ and the induction hypothesis can be applied to them, obtaining $\theta(\mathrm{sym}\ A\ c\ a\ \pi_2) : a =_A c$ and $\theta(\mathrm{sym}\ A\ b\ c\ \pi_1) : c =_A b$. From that we obtain

$$(\mathrm{trans}\ A\ a\ c\ b\ \theta(\mathrm{sym}\ A\ c\ a\ \pi_2)\ \theta(\mathrm{sym}\ A\ b\ c\ \pi_1)) : a =_A b$$

Second case We know that (sym A b a (sym A a b π)) : $a =_A b$, thus (sym A a b π) : $b =_A a$ and $\pi : a =_A b$. Induction hypothesis suffices to prove $\theta(\pi) : a =_A b$

Third case Since (trans A a b b π_1 π_2) : $a =_A b$ we have $\pi_1 : a =_A b$. Again, the induction hypothesis suffices to prove $\theta(\pi_1) : a =_A b$

Fourth case Analogous to the third case

Fifth case By hypothesis we know that

$$(\text{sym } B\ \Delta[a]\ \Delta[b]\ (\text{eq\_f } A\ B\ \Delta\ a\ b\ \pi)) : \Delta[b] =_B \Delta[a]$$

Thus $\pi : a =_A b$ and (eq_f A B Δ a b π) : $\Delta[a] =_B \Delta[b]$. Hence (sym A a b π) : $b =_A a$ and

$$(\text{eq\_f } A\ B\ \Delta\ b\ a\ (\text{sym } A\ a\ b\ \pi)) : \Delta[b] =_B \Delta[a]$$

Sixth case Follows directly from the inductive hypothesis

□

5 Conclusion and related works

In this paper we have presented a procedure to transform a minimal proof trace left by an automatic proof searching procedure to a valuable proof term in the calculus of inductive constructions. We then refined this proof object with type preserving transformations, making it suitable for the natural language rendering engine of the Matita interactive theorem prover.

The problem of reconstructing a proof from some sort of trace left by an automatic prover is addressed by Hurd in [5] and by Kreitz and Schmitt in [6] while developing JProver[16]. In the former work, Hurd has to face the problem of reconstructing a proof from the ambiguous and incomplete output of the Gandalf[19] prover, and he solves it inferring the missing information with a prolog-style search. On the contrary, when we wrote the automatic procedure we had in mind that the output would have been a formal proof, thus we paid attention in not trading the proof trace completeness down for efficiency. The latter work describes several proof reconstruction methodologies in order to obtain natural deduction style or sequent style proofs from resolution and matrix based proof traces. Since we restricted our automatic procedure to the unit equality case, we do not have real clauses and we implement only a trivial subset of the resolution calculus with the equality resolution rule, thus these approaches do not fit well in our setting.

There is a wide literature on the integration of automated procedures with interactive provers, but they usually focus on slightly different aspects or drop some of the requirements we consider essential, anyway they give good suggestions on possible improvements of our work. Meng and Paulson were interested in integrating one of the best ATP systems, Vampire[14], with Isabelle[10] and studied a set of transformations[9, 8] to encode (fragments of) the expressive HOL logic into the first order one implemented by Vampire. Some of these techniques could be applied in our case too, allowing us to treat a larger fragment of CIC with our automatic procedure. Ayache and Filliâtre have integrated many ATP systems, like haRVey[4] and CVC Lite[3] with the Coq[20] interactive theorem prover, encoding a fragment of the logic of Coq (CIC) into the intermediate polymorphic first order logic[2] (PFOL) logic, which is meant to be easily convertible to the logics understood by the ATP systems. While the translation to PFOL could be relevant for future improvements of our work, the rest of the paper drops the requirement of producing proof objects, trusting the essentially boolean answer of the ATP systems. Matita follows the De Bruijn principle, stating that proofs generated by the system should be verifiable with a small tool, and since in general an ATP system cannot be considered small, we consider the generation of a proof object that can be verified with a small kernel mandatory. Consider for example that haRVey counts nearly 50,000 lines of code and CVC Lite more then 70,000 while the kernel (type checker) of Matita only 10,000.

The main distinctive characteristic of our work is in the way we take care of the proofs we found; first encoding them in a formal calculus, then improving them both from a practical (space/type-checking efficiency) and an esthetical (natural language rendering) point of view. As suggested above, a natural continuation of this work would be to study how to treat a bigger fragment of CIC with the automatic procedure we implemented without dropping the fundamental requirement of being able to exhibit a valuable CIC proof term once a proof is automatically found.

References

1. A. Asperti and C. Sacerdoti Coen and E. Tassi and S. Zacchiroli. *User Interaction with the Matita Proof Assistant.* Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving. To appear.
2. N. Ayache and J.C. Filliâtre. *Combining the Coq proof assistant with first-order decision procedures.* Unpublished.
3. C. Barrett and S. Berezin. *CVC Lite: A New Implementation of the Cooperating Validity Checker.* Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04), LNCS Vol. 3114, 2004, 515-518.
4. D. Déharbe and S. Ranise and P. Fontaine. *haRVey, a cocktail of theories.* http://harvey.loria.fr/
5. J. Hurd. *Integrating Gandalf and HOL.* Proceedings of Theorem Proving in Higher Order Logics (TPHOL) 1999, 311-321
6. C. Kreitz and S. Schmitt. *A Uniform Procedure for Converting Matrix Proofs into Sequent-Style Systems.* Information and Computation, Vol. 162(1-2), 226-254, 2000
7. W. Mc Cune. *Solution of the Robbins Problem.* Journal of Automated Reasoning, Vol. 19(3), (1997) 263-276.
8. J. Meng and L. Paulson. *Experiments On Supporting Interactive Proof Using Resolution.* In: David Basin and Michael Rusinowitch (editors), IJCAR 2004

9. J. Meng and L. Paulson. *Translating Higher-Order Problems to First-Order Clauses.* Geoff Sutcliffe, Renate Schmidt and Stephan Schulz (editors), ESCoR: Empirically Successful Computerized Reasoning (CEUR Workshop Proceedings, Vol. 192, 2006), 70-80.
10. J. Meng, C. Quigley and L. Paulson. *Automation for Interactive Proof: First Prototype.* Information and Computation, Vol. 204(10), 2006, 1575-1596.
11. R. Nieuwenhuis and A. Rubio. *Paramodulation-Based Theorem Proving.* Handbook of Automated Reasoning (2001) 371-443.
12. S. Obua and S.Skalberg. *Importing HOL into Isabelle/HOL.* In Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR), LNAI Vol. 4130, (2006) 298-302.
13. C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Suprieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996,
14. A. Riazanov and A. Voronkov. *The design and implementation of VAMPIRE.* AI Communications, Vol. 15(2-3), (2002) 91-110.
15. A. Riazanov. *Implementing an Efficient Theorem Prover.* PHD thesis, The University of Manchester, 2003.
16. S. Schmitt and L. Lorigo and C. Kreitz and A. Nogin. *Jprover: Integrating connection-based theorem proving into interactive proof assistants.* International Joint Conference on Automated Reasoning, LNAI, Vol. 2083, 2001, 421-426.
17. J. Slaney and M. Fujita and M. Stickel. *Automated Reasoning and Exhaustive Search: Quasigroup Existence Problems.* Computers and Mathematics with Applications, 1993
18. G. Sutcliffe and C.B. Suttner. *The TPTP Problem Library: CNF Release v1.2.1.* Journal of Automated Reasoning, Vol. 21(2), (1998) 177-203.
19. T. Tammet. *A resolution theorem prover for intuitionistic logic.* Proceedings of the International Conference on Automated Deduction, LNAI, Vol. 1104, 1996.
20. The Coq Development Team: *The Coq Proof Assistant Reference Manual.* http://coq.inria.fr/doc/main.html (2006).
21. B. Werner. *Une Théorie des Constructions Inductives.* PHD thesis, Université Paris VII, 1994.