Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 226 (2009) 89–105
www.elsevier.com/locate/entcs

An Interactive Driver for Goal-directed Proof Strategies

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

Interactive Theorem Provers (ITPs) are tools meant to assist the user during the formal development of mathematics. Automatic proof searching procedures are a desirable aid, and most ITPs supply the user with an extensive set of facilities to improve automation. However, the black-box nature of most automatic procedure conflicts with the interactive nature of these tools: a newcomer running an automatic procedure learns nothing by its execution (especially in case of failure), and a trained user has no opportunities to interactively guide the procedure towards the solution, e.g. pruning wrong or not promising branches of the search tree. In this paper we discuss the implementation of the resolution based automatic procedure of the Matita ITP, explicitly conceived to be interactively driven by the user through a suitable, simple graphical interface.

Keywords: Interactive theorem proving, SLD resolution, automation

1 Introduction

Most of the development effort behind Interactive Theorem Provers is devoted to bridge the gap between the high level language used by humans for reasoning and communicating mathematics, and the low level foundational language understood by ITPs. Among all facilities offered by ITPs, a high degree of automation is certainly desirable and several works (see for example [12,11]) have been devoted to the integration of automatic proof search facilities in interactive theorem provers. The machinery employed in this integration is usually hidden to the user: when the automatic procedure finds a proof the interactive theorem prover usually evaluates the trace left by the prover (if any) and converts it, possibly using some reflection mechanism (see [5,6]), to

doi:10.1016/j.entcs.2008.12.099

a proof in its foundational dialect. What is neglected by this traditional approach is the interactive nature of the tool. The user has no feeling of what is going on, why the automatic procedure has possibly failed and how he can possibly improve the situation. Moreover, when used in a didactical environment where untrained users are put in front of an interactive theorem prover, it is desirable to let them use automation facilities freely, but providing them the possibility to understand the work done by the automatic procedure or the reasons of its failure.

The aim of this work is to develop a reasonably fast SLD [13,14] based proof searching procedure for the interactive theorem prover Matita [3] that is completely transparent to the user, allowing him to follow the execution of the procedure and to drive it, taking run-time decisions on how the procedures explores the search space. As a side effect we obtain a very handy debugging tool, that proved to be extremely useful to tune and fix the procedure.

To get this result, we develop a SLD engine that performs backtracking without relying on the call stack (i.e. not using stack frames as choice points). This characteristic, together with a carefully chosen selection function, allow us to effectively present to the user a view of the ongoing computation.

2 The proof searching procedure

The way proofs are built in Matita is by instantiation. The foundational dialect of the interactive theorem prover (namely the Calculus of Inductive Constructions [9,16]) is extended with meta-variables [15] (written $?_i$) whose type represents a missing part of the proof, called *goal*.

Definition 2.1 [Proof problem] A *proof problem* $\mathcal{P}$ is a finite list of typing judgement of the form $\Gamma \vdash ?_j : T$ where for each metavariable $?_i$ that occurs in the context Γ and type T there exists a corresponding entry in $\mathcal{P}$.

Each proof step generates a substitution instantiating one or more existing metavariables, whose entries are also removed from $\mathcal{P}$, and possibly adding new entries (new open goals) to $\mathcal{P}$.

Definition 2.2 [Substitution] A metavariable substitution Σ is a list of couples metavariable-term.

$$\Sigma = [?_1 := t_1; \ldots; ?_n := t_n]$$

Substitutions are usually performed lazily, thus the status of the ongoing proof comprises both a proof problem and a substitution. We will call such a pair a *proof status*.

For example, the initial status of the just declared conjecture $\forall x, y : \mathbb{N}.P(x, y) \to Q(x, y)$ will be

$$[] \vdash ?_1 : \forall x, y : \mathbb{N}.P(x, y) \to Q(x, y)$$

together with an empty substitution. After performing hypothesis introduction it will change to

$$x, y : \mathbb{N};\ p :\ P(x, y) \vdash ?_2 : Q(x, y)$$

together with a substitution $\Sigma = [?_1 := \lambda x, y : \mathbb{N}.\lambda p : P(x, y).?_2]$.

The application of a substitution Σ to a term t is denoted with $\Sigma(t)$. This operation is extended to contexts and proof problems, substituting all the types of abstracted variables (in the context) or the types of of missing proofs (in the proof problem).

A proof is over when there are no more proof problems in the proof status, and the proof of the original conjecture can be obtained applying the substitution to the initial metavariable.

The proof searching procedure we implemented in the interactive theorem prover Matita is essentially inspired by SLD resolution [14]: it iterates applications of known results following a depth-first strategy (up to a given depth). No introduction of new hypothesis is done (that amounts to assume to have a horn-like base of knowledge, as it is often the cases), hence the context of the proof remains unchanged during the execution of the procedure.

The classical rule for SLD resolution follows.

SLD

$$\frac{\leftarrow A_1, \ldots, A_n \qquad H \leftarrow B_1, \ldots, B_m \qquad \Sigma = mgu(H, A_i)}{\leftarrow \Sigma(A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)}$$

CIC is a dependently typed, higher order, language where no most general unifier can be found in the general case. Nevertheless, an essentially first order unification heuristic is implemented as part of the so called *refiner*[1] and largely used in the process of building proofs. A detailed description of the unification algorithm implemented in Matita can be found in [18] and some recent extensions are described in [19].

[1] The *refiner* is the component implementing type-inference, as opposed to the *kernel*, implementing type-checking. It is in charge to automatically fill the proof with a lot of negligible information easily inferred by the context. See e.g. [2] for an architectural outline of Curry-Howard based ITPs.

Definition 2.3 [Unification] The process of unifying two terms is denoted with

$$\mathcal{P},\ \Sigma,\ \Gamma \vdash N \stackrel{?}{\equiv} M \ \stackrel{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma'$$

Unification performs only metavariables instantiations, and the resulting Σ' is such that $\Sigma'(N)$ is convertible (that for CIC means equal up to $\beta\iota\delta\zeta$-reduction) with $\Sigma'(M)$ in context $\Sigma'(\Gamma)$ and proof problem $\Sigma'(\mathcal{P}')$.

The SLD resolution rule is implemented in Matita as the *apply* tactic. Since it is meant for interactive usage, both the selection and computation rule are left to the user: in the following presentation the goal i and the clause (lemma) c are user provided. The outcome of the tactic is a proof status or an exception if the unification step fails.

Apply-tac

$$\frac{\begin{array}{l}\mathcal{P} = \Gamma_1 \vdash ?_1 : A_1, \ldots, \Gamma_n \vdash ?n : A_n \\ \mathcal{P}' = \mathcal{R}(\Gamma \vdash ?_{B_1} : B_1, \ldots \Gamma, x_1 : B_1, \ldots, x_{m-1} : B_{m-1} \vdash ?_{B_m} : B_m); \mathcal{P} \\ \Gamma \vdash c\ ?_{B_1}\ \ldots\ ?_{B_m} : H \\ \mathcal{P}',\ \Sigma,\ \Gamma \vdash H \stackrel{?}{\equiv} A_i \ \stackrel{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}'',\ \Sigma' \\ \Sigma'' = ?_i := c\ ?_{B_1}\ \ldots\ ?_{B_m}; \Sigma'\end{array}}{(\mathcal{P}'', \Sigma'')}$$

With $\Gamma \vdash t : T$ we denote the typing judgement assigning to t the type T in the context Γ. The reordering function $\mathcal{R}$ is applied to the list of new goals, and as we will see in Section 2.1 it allows to implement some heuristics to increase performances and avoid the proliferation of meaningless goals.

Note that unifying H with A_i can in general instantiate some $?_{B_i}$ but not generate new metavariables, thus the set of new goals opened by the apply tactic is a subset of $\{?_{B_1}, \ldots, ?_{B_n}\}$.

Our final goal is to provide the user a tool to observe the automatic procedure running and possibly drive it without stopping it. To do that, we have to make sure that some parts of the computation are reasonably stable, such that the user has enough time to read them before they change. If it was not possible, the user would have to stop the execution and make it advance step by step, inherently loosing the speed modern computers have, or alternatively not use the tactic interactively (just let it run).

To achieve a reasonably stable view of the ongoing computation, we had to adopt a leftmost, depth first, selection rule. The selection function is fixed and always chooses the first goal, in the same spirit of Prolog. The proof the

procedure is building up can be seen as some sort of tree: an application of the resolution rule generates a node with a new son for every newly generated goal, and proceeds trying to prove all of them. If one fails it backtracks changing the node (if there are alternative clauses that can be applied). If we assume to have n applicable clauses and a depth limit d, a node at depth i is updated every $(d-i-1)^n$ iterations, granting a reasonable stability for shallow nodes.

An alternative search strategy, like for example the discount algorithm [17], that generates and continuously refines a set of proved (intermediate) results, would not have worked. What a user needs to know to understand what a discount based automatic prover is doing is the set of intermediate lemmas proved so far. This set is usually really huge and continuously changing: new results are added, weaker results are removed in favour of more general ones, all results are simplified (put in a canonical form) using newly generated equations.

2.1 The reordering function

To understand why reordering newly generated goals can increase performances, and also avoids generating many pointless goals, consider the division operation between natural numbers and the associated predicate *divides*. A natural number q divides n if there exists a p such that $n = q * p$. In a dependently typed λ-calculus equipped with inductive types, a natural [2] definition for that predicate would be an inductive predicate with a single constructor

$$witness : \forall p, q, n : \mathbb{N}.p * q = n \rightarrow q|n$$

This lemma (actually a constructor), when applied, generates two new goals: $?_p$ of type $\mathbb{N}$ and $?_H$ of type $?_p * q = n$. Attempting to solve $?_p$ first is a bad idea since we have no real information on $?_p$ except that it is a natural number, while we know more information concerning the second goal, for example that it involves the multiplication operation. This piece of information can be exploited by the computational rule to search for applicable clauses. Moreover, almost every solution to goal $?_H$ also forces $?_p$ to be some fixed natural number.

Interactive theorem provers are tools used to create libraries of formalized theorems; as a consequence the environment from which the computation rule may choose a lemma to apply is extremely polluted. In case of goals of just type $\mathbb{N}$, it could even choose to apply the Fibonacci function and then successively try to guess an input such that the second goal can be solved,

[2] An alternative definition, using the computational fragment of CIC to define the division operation and proving some properties of that function is also possible, but not widely adopted.

possibly backtracking and guessing another input for the Fibonacci function. The ability of the CIC logic to compute is very handy in general, but is cases like this one may lead to very long computations.

2.2 The computation rule

The computation rule has to find a clause (in our case an existing lemma), or better a list of clauses, that will be applied in order to solve a given goal. ITPs are equipped with large libraries of already proved results, thus some searching facilities have to be employed to select a reasonably small amount of lemmas that will then be effectively applied. Matita has many built-in searching facilities, extensively described in [1], that can search local and remote libraries for results relevant to a given goal. These facilities are used to fill in an in-memory trie[3] data structure together with some parts of the library the user can declare to be pertinent to what he is doing. On top of this structure a pretty efficient unification approximation can be performed, resulting in a set of lemmas that is later refined using the real unification algorithm.

Since we want to present the user only good alternatives, the computational rule has not only to find good candidates, but also to attempt to apply them, directly pruning false positives. Moreover, suddenly applying all found lemmas allows to sort these alternatives looking for example to the number of newly opened goals. The cands function performs this search and returns a list of alternative proof statuses.

Definition 2.4 [Candidates (of the environment E)] Let g be a goal, $\mathcal{P}$ a proof problem and Σ a substitution environment. Let $\Gamma \vdash ?g : T \in \mathcal{P}$. The function cands applied to a proof status ($\mathcal{P}$, Σ) and a goal g returns a list of tuples $(\Sigma', \mathcal{P}', [g_1; \ldots; g_n])$ such that:

- $t \in E$
- $\Gamma \vdash t : \forall x_1 : T_1. \ldots \forall x_n : T_n.T'$
- $\mathcal{P},\ \Sigma,\ \Gamma \vdash T \stackrel{?}{\equiv} T' \ \stackrel{\mathcal{U}}{\rightsquigarrow}\ \mathcal{P}',\ \Sigma'$
- $\Gamma; x_1 : T_1; \ldots; x_{i-1} : T_{i-1} \vdash ?g_i : T_i \in \mathcal{P}' \quad \forall i \in \{1, \ldots, n\}$
- $?_g := (t\ ?g_1\ \ldots\ ?g_n) \in \Sigma'$

2.3 Backtracking

The cands function finds a set of relevant lemmas in the global environment (the library of already proved results) and using the **Apply-tac** rule attempts

[3] A trie is a tree of prefixes, a good compromise between search speed and space consumption adopted, in some of its variants, by many automatic provers.

to apply them to a given goal, returning the list of proof statuses relative to successful applications of that rule. On top of that, an automatic proof searching procedure can easily be implemented by means of two mutually recursive functions.

For each goal to be solved (gl), the function *search* calls the computation rule (implemented by the cands function) that finds a list of lemmas and that uses the **Apply-tac** rule to obtain the list of associated proof statuses (cl). Then it tries to find if one of the resulting proof statuses can be solved, using the *first* function, that recursively calls *search*. If one succeeds, *search* moves to the next goal to be solved. A pseudo-OCaml code for that function follows. The choice of OCaml as the implementation language for the tactic is not arbitrary, since the whole Matita ITP is written in in that language.

```
let rec first f l = function
  | [] → raise Failure
  | hd::tl →
      try f hd
      with Failure → first f tl
let rec search gl (S, P) =
  match gl with
  | [] → S, P
  | g::tl →
      let cl = cands (S, P) g in
      let S',P' = first (fun (S, P, gl) → search gl (S, P)) cl in
      search tl (S', P')
```

The code is oversimplified, many checks are missing: for example there is no bound check, thus this function may diverge. Nevertheless, it is already enough to see the issue arising with this simple and elegant implementation of backtracking.

The problem with this approach is that informations needed to properly backtrack are kept by the OCaml stack. The try/with construct uses stack frames to "label" choice points in the derivation to which the function may backtrack. While this is in general an elegant solution, it can not be employed here, since we want to show the user the current computation, and OCaml (like most of compiled languages) does not provide enough introspection mechanisms to explore the current call stack.

To reach our objective we have to write a stack-less procedure (that is a tail recursive function). Before detailing such procedure we want to give an overview of the final result we obtained, showing the interface we offer to the user.

3 The graphical user interface

The proof searching procedure elaborates fast, but the depth-first proof searching strategy (that is, selecting always the first goal) makes the shallow part of the computation pretty stable. For that reason we adopted the viewport widget, that allows to display only a subpart of a larger picture, by default the most stable.

In Figure 1 the user interface to drive the automatic procedure is shown. On the background there is the main window of Matita, showing the current open conjecture (conjecture fifteen). The window is divided in three columns:

- the leftmost shows the progressive number of open conjectures, the number identifying the current goal and the depth left (the difference between the user defined bound and the actual depth);
- the column in the middle displays the i-th open conjecture, since it lives in the original context (displayed by the background window) there is no need to print again this information;
- the rightmost column lists all lemmas that can be applied to the conjecture. This column displays the so called choice stack [7], colouring in grey the applied lemma. Some additional information on these lemmas are displayed using tool tips. If a lemma is unknown to the user, its type can be shown holding the mouse on its name.

To attack conjecture fifteen the automatic tactic found a bunch of lemmas that can be applied. The former, witness, has already been applied and is thus coloured in grey. The list of grey items, read top to bottom, is the list of lemmas applied so far. All its alternatives are shown on its right. The application of the witness lemma to a goal of the form $n|m$ opens two conjectures: the former (number 52) is that for a certain $?_{51}$, $m = n*?_{51}$ and the latter (number 51) is the witness $?_{51}$ itself.

The user already sees the result of the reordering function $\mathcal{R}$, since newly opened goals have been sorted, preferring goal 52 to 51.

The next step performed by the automatic procedure is to find relevant lemmas for the conjecture displayed in the second line, place them in the rightmost column, grey the former and display the result of its application. In case one application fails, the next alternative is attempted. In case there are no alternatives left, the next alternative of the previous line it considered. Thus, if no lemmas can be applied to conjecture 52, both line one and two are removed together with the witness lemma that generated them and the lemma div_mod_spec_to_divides is applied.

The user can execute the tactic step by step with the next button, and

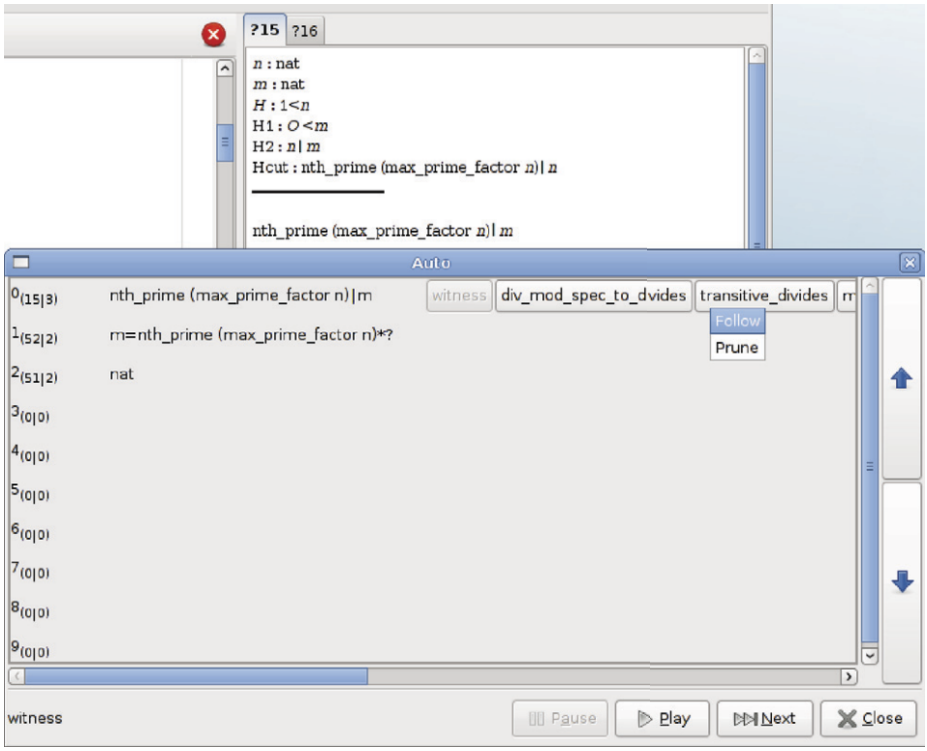

Figure 1. Auto interaction window

switch between the running status and the paused one with the buttons pause and play. To drive the proof searching algorithm the user can interact with the lemmas in the rightmost column. In Figure 1 the user just clicked on the transitive_divides lemma, opening the list of allowed actions. The *prune* action simply removes the lemma for the list of alternatives, the *follow* action makes all alternatives before the one selected immediately fail.

The pair of big arrow buttons on the right allows to move the current viewport, focusing on goals that are examined by the proof searching procedure at depth greater than a fixed amount (ten in this case) in the search tree. The choice of using a viewport allows to cut out the deepest part of the computation, that is likely to change very frequently and not worth being displayed.

When a subgoal is solved, two possible scenario arise, depending if some metavariables are occurring in its statement or not. If some metavariables occur, the solution found may instantiate them in such a way that other goals in which such metavariables occur result false. In that case, the line corresponding to that goal is not removed, and the list of candidates associated to it remains visible and the user can interact with it. If the goal statement

contains no metavariables the corresponding line is removed, since no choices relevant for the eventual success of the proof search procedure can be made by the user.

4 Operational description of the tactic

To present the user such a window, the search procedure has to be stackless. All informations have to be accessible by the graphical user interface at any time. That means the procedure has to be a for-program (or a tail recursive function) keeping the computation tree (and informations needed for backtracking) into a first order object and possibly pass it to the GUI.

To formally describe how the procedure works and the data structure used to represent the computation status we need to define the following objects.

Definition 4.1 [Proof of goal] Given a goal (metavariable number) g and a substitution Σ, the proof of g denoted with $\Sigma(g)$ is the least fixed point of $\Sigma(\cdot)$ starting from $?g$.

This function is not only used at the end of the tactic to build the proof object for the main conjecture, but also to create (and cache) the proof of intermediate results, avoiding to search twice the same proof.

Definition 4.2 [Metas of term] Given a term t the set of metavariables occurring in t is denoted with $\mathcal{M}(t)$.

As we already anticipated in the previous section, the procedure behaves differently if a metavariable occurs in a goal.

Definition 4.3 [Cache] A cache θ is a partial function from terms (actually types) to terms. Its domain can be extended with the operation $\theta[T \mapsto t]$. All terms in θ live in the same context.

We use the notation $\theta[T \mapsto \Sigma(g)]$ to update θ associating the proof of g with T. We use $\bot$ to represent failures, thus $\theta[T \mapsto \bot]$ extends θ with the information that T has no proof. The cache is an essential ingredient to obtain good performances and avoids many kinds of loops.

Definition 4.4 [Element] We call an element a triple of type (in OCaml notation) proof status $*$ op list $*$ goal list where goal is the type of metavariable indexes and op is the following algebraic type:

```
type op = D of goal | S of goal * term
```

The D constructor will decorate goals that still have to be processed (toDo), while S will decorate goals that have been successfully solved, and whose proof may be cached. The last component of an element is a failure list, containing all goals that have to be considered failed when the element itself fails (i.e. when the *op* list contains some D items that fail).

The last ingredient is the function to find lemmas that can be applied to a given goal, that is the function cands described in Section 2.2. The only needed modification is to make this function also return the applied lemma together with the proof status: this is needed to display the choice stack to the user. Note that cands can easily be extended to look for applicable lemmas not only in the global environment E but also in θ since all elements in θ live in the same context Γ of the goal (the proof searching procedure never alters Γ).

In Table 1 we define the *step* function mapping a list of elements and a cache to a new list of elements equipped with a possibly updated cache. This function is the core of the automatic procedure, and is applied until a Failure or Success status is reached. We use $\circ$ for list concatenation. The complete failure status is represented by $([], \theta)$: the elements list can be considered to list all the alternatives that can be used prove the initial goal, being empty means that all alternatives have been explored with a negative result. The annotation t in S_g^t is not used in the operational semantic, and t represents the lemma that was applied to g. Remember we have to show the user the history of lemmas applied so far. The procedure starts with the following configuration, where g is the initial goal and P the initial proof status and θ an empty cache.

$$([(P, [D_g], [])], \theta)$$

On such a status the step function applies rule (vi). calling the cands function to get a list of alternative proof statuses. All new goals are decorated with a D constructor, and sorted using the $\mathcal{R}$ function. They are positioned in front of the tl list, separated with an S item for the processed goal g. This item, when processed, will cache the proof found for g, and this will happen only after all newly created D items are solved.

In our example, assuming the result of the cads function amounts to $cands(P, g) = [(t_1, P_1, [g_1]);\ (t_2, P_2, [g_2; g_3])]$ we obtain the following state.

$$([(P, [D_g], [])], \theta) \xrightarrow{step} ([(P_1, [D_{g_1}; S_g^{t_1}], []);\ (P_2, [D_{g_2}; D_{g_3}; S_g^{t_2}], [g])], \theta)$$

Note that a new element is generated for every alternative proof status returned by the cands function. All of them, except the last one, are equipped

$$(((\mathcal{P},\Sigma) \text{ as } P, S^t_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, tl, fl) :: el', \theta') \quad \text{(i)}$$
when $\mathcal{M}(T) = \emptyset$ and $\Gamma \vdash ?g : T \in \mathcal{P}$
where $\theta' = \theta[T \mapsto \Sigma(g)]$ and $el' = purge(el, tl)$

$$(((\mathcal{P},\Sigma) \text{ as } P, S^t_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P, tl, fl) :: el, \theta) \quad \text{(ii)}$$
when $\mathcal{M}(T) \neq \emptyset$ and $\Gamma \vdash ?g : T \in \mathcal{P}$

$$(((\mathcal{P},\Sigma), D_g :: tl, fl) :: el, \theta) \xrightarrow{step} (((\mathcal{P},\Sigma'), tl, fl) :: el, \theta) \quad \text{(iii)}$$
when $\theta(T) \neq \bot$ and $\Gamma \vdash ?g : T \in \mathcal{P}$
where $\Sigma' = \Sigma \circ [?g := \theta(T)]$

$$(((\mathcal{P},\Sigma), D_g :: tl, fl) :: el, \theta) \xrightarrow{step} (el, \theta'_{m+1}) \quad \text{(iv)}$$
when $\theta(T) = \bot$ and $\Gamma \vdash ?g : T \in \mathcal{P}$
where $\theta'_1 = \theta$ and $fl = \{g_1; \ldots; g_m\}$
and $\Gamma_g \vdash ?g : T_g \in \mathcal{P}$ for $g \in \{1, \ldots, m\}$
and $\theta'_{g+1} = \theta'_g[T_g \mapsto \bot]$ for $g \in \{1, \ldots, m\}$

$$(((\mathcal{P},\Sigma), D_g :: tl, fl) :: el, \theta) \xrightarrow{step} (el, \theta'_{m+1}) \quad \text{(v)}$$
when $cands(P, g) = []$
where $\theta'_1 = \theta$ and $fl = \{g_1; \ldots; g_m\}$
and $\Gamma_g \vdash ?g : T_g \in \mathcal{P}$ for $g \in \{1, \ldots, m\}$
and $\theta'_{g+1} = \theta'_g[T_g \mapsto \bot]$ for $g \in \{1, \ldots, m\}$

$$((P, D_g :: tl, fl) :: el, \theta) \xrightarrow{step} ((P'_1, l_1@tl, []) :: \ldots :: (P'_m, l_m@tl, g :: fl) :: el, \theta) \quad \text{(vi)}$$
where $cands(P, g) = (t_1, P'_1, g_{1,1} \ldots g_{1,n_i}) :: \ldots :: (t_m, P'_m, g_{m,1} :: \ldots :: g_{m,n_m})$
and $l_i = \mathcal{R}([D_{g_{i,1}} \ldots; D_{g_{i,n_i}}]) \circ [S^{t_i}_g]$ for $i \in \{1 \ldots m\}$

$$((P, [S^t_g], fl) :: el, \theta) \xrightarrow{step} \text{(Success P)} \quad \text{(vii)}$$

$$([], \theta) \xrightarrow{step} \text{Failure} \quad \text{(viii)}$$

Table 1
Automatic procedure operational description

with an empty failure (fl) list. In that way, if they fail, the cache will not be updated with a failure for g, since there are still valid alternatives for that goal. On the contrary, the last element inherits the failure list and adds to it g.

Rules (i) and (ii) process a success (that is an S item). The first rule is applied when no metavariable occurs in the goal, thus the proof found will not have side effects on the rest of the computation and can be safely added to the cache θ. In that case, the purge function is used to drop alternatives (brothers of g). They can be identified in the flat el list comparing the list

of items, since the tl is inherited by all brothers (in rule (vi)) and is never modified.

Rule (iii) solves a D_g item when the cache θ holds a proof for the goal g. The substitution is enriched with an entry for g.

Rules (iv) and (v) are for partial failures, the former is applied when no applicable clauses are found, the latter when a failure was previously cached for the same goal.

Rule (vii) is for success, that is when no more items have to be processed. The final proof status is returned.

4.1 Improvements

The procedure presented in Table 1 can be improved in many ways, for example giving a bound to the search space or refining the caching mechanism. These improvements have been omitted from Table 1 to increase its readability, but are explained in the following.

To limit the search tree explored by the procedure to a certain depth, or even a number of nodes, some additional fields have to be added to the element structure. To efficiently keep track of the depth or size of the tree, the element structure is enriched with two integers representing the depth left and the actual size of tree: every time a D item is processed, the depth limit (as well as the size) is decreased. When an S item is processed the depth is increased again. The additional following rule is then added to the operational description:

$$((P, items, fl, depth, size) :: el, \theta) \xrightarrow{step} (el, \theta) \quad \text{(iii bis)}$$
$$\text{when } depth < 0 \vee size < 0$$

The cache θ is still not optimal, since a goal g of type T can be associated with $\bot$ because the algorithm run out of depth (or size). If the algorithm encounters again the same goal type T with a greater depth, it could retry. To fix this problem, goals have to be paired with the depth at which they have been generated in the failure (fl) list, and the $\bot$ symbol annotated with that

depth. Then rule (iv) can be refined as follows:

$$
\begin{aligned}
&(((\mathcal{P},\Sigma) \text{ as } P, D_g :: tl, fl, depth, size) :: el, \theta) \xrightarrow{step} (el, \theta'_{m+1}) \qquad \text{(iv)}\\
&\quad \text{when } \theta(T) = \bot_k \text{ and } k \geq depth \text{ and } \Gamma \vdash ?g : T \in \mathcal{P}\\
&\quad \text{where } \theta'_1 = \theta \text{ and } fl = \{(g_1, d_1); \ldots; (g_m, d_m)\}\\
&\quad \text{and } \Gamma_g \vdash ?g : T_g \in \mathcal{P} \text{ for } g \in \{1, \ldots, m\}\\
&\quad \text{and } \theta'_{g+1} = \theta'_g[T_g \mapsto \bot_{d_g}] \text{ for } g \in \{1, \ldots, m\}
\end{aligned}
$$

Note that the last line stores failures for goals in the fl list that have to be enriched with the depth at which they have been processed in rule (vi).

The cands function can be modified to properly sort the list of returned proof statuses, in such a way that the most promising ones are processed first. The simplest heuristic is to count the number of newly generated goals (the length of l_i in rule (iv)).

4.2 Interfacing with the GUI

The GUI and the automatic procedure run in different threads. Rule (vi) checks a condition variable[4], associated with the *pause* button of the GUI, before proceeding. The computation status (the el list) is purely functional and every loop sets a global reference to that variable, allowing the GUI thread to render it.

The element list contains all the information needed by the GUI, but not in an handy format. The automatic procedure and the data structure it manipulates have been designed with both speed and user friendliness in mind, but execution speed has been always preferred to rendering speed or to making the rendering process easier. The function to map the element list into a data structure suitable for the GUI is not interesting, even if far from being trivial, and will not be detailed here. It essentially amounts in processing in parallel all *op* lists (one for every element in the el list), grouping together the lemmas stored in S items. The lemma recorded in S items is shown to the user as the choice made the procedure. The actual statements of goals can be computed using the proof status $P = (\mathcal{P}, \Sigma)$, since all goals have an entry in the proof problem $\mathcal{P}$, and eventual instantiations of metavariables occurring in their types is recorded in the substitution Σ.

[4] A condition variable is a widespread synchronisation mechanism allowing one execution context to wait for a boolean variable to became true, and another execution context to change the value of that variable eventually waking up every thread waiting on that variable.

5 Related works

Many debugger or trace visualisation tools have been proposed by the logic/-constraint programming community. Most of them like the ones described in [21,8] fall in the so called post-mortem trace analyser, allowing the user to inspect the computation once it has terminated.

The recent CLPgui [10] employs 2D and 3D visualisation paradigms to show the user the full search tree, allowing him to navigate it and zoom the interesting parts of the computation trace.

OzExplorer [20] adopts subtree folding to make the whole tree fit the screen, a requisite we do not have and thus we adopt a simpler viewport (a restricted view of the search tree). Moreover we hide solved subgoals (when their solution is not a choice, i.e. they do not instantiate any metavariable present in any other goal). [7] introduces the notion of choice stack (list of choices made so far), similar to our list of grey buttons in the rightmost column.

While our work shares some ideas and follows some visualisation paradigms described in these papers, the use case of our procedure in an ITP is clearly different from the general use case of a CLP program. These differences are summarised in the following:

- our GUI is rarely used to display a huge program (computation), thus it is tailored to the most frequent case of a tree of depth less then ten
- in ITPs like Matita, thanks to the reasonably large library that equips them, the branching factor is very high and that prevents a proper tree display: siblings would be too far to be visually related, thus we dropped the idea of visualising a tree
- every goal has a meaning per se, thus many informations like goals already solved can be hidden. The choice stack tells the user where the goal comes from and this information is enough to follow the computation

For these reasons we had to develop a novel user interface, instead of reusing or adapting one of the aforementioned tools.

6 Conclusions

In this paper we presented a SLD resolution based automatic procedure for the interactive theorem prover Matita, that is designed to be driven by the user through a graphical user interface. In this way we allow unexperienced user to observe the procedure running, possibly understanding why it fails or how it managed to solve a goal for them. Trained users can easily tune the procedure pruning not promising branches of the computation or following

good ones.

A still work in progress addition to this work is making the procedure generate not only a proof object, but also a proof script (the list of primitive commands to generate the proofs object) in the spirit of [4]. The choices made by the user interactively have to be recorded so that running again the automatic procedure possibly honours the same user requests. Having a proof script does not only show the user what the procedure did, but also greatly decreases the amount of time needed to re-check the proof script (since proof search has not to be performed again). Formalising mathematics with an ITP is not an easy task, and refining definitions is a really frequent activity that usually breaks many already proved lemmas. Having just a call to an automatic procedure can slow down the process of mending broken proof scripts, especially if there is no way to inspect what the procedure does, making it harder to understand the reasons of a failure. Our work already ameliorates this situation, but having a proof script that details the previously found proof, would be even better, allowing a fast re-execution and detection of the problem, and allowing the user to fix the proof directly if possible, or re-run the automatic procedure driving it towards a working proof.

References

[1] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Post-proceedings of the Types 2004 International Conference*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2004.

[2] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *Proceedings of Types 2006: Conference of the Types Project. Nottingham, UK – April 18-21*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2007. Special Issue on User Interfaces for Theorem Proving. To appear.

[4] Andrea Asperti and Enrico Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In *Calculemus/MKM*, pages 146–160, 2007.

[5] Gilles Barthe, Mark Ruys, and Henk Barendregt. A two-level approach towards lean proof-checking. In *Types for Proofs and Programs (Types 1995)*, volume 1158 of *LNCS*, pages 16–35. Springer-Verlag, 1995.

[6] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 515–529. Springer-Verlag, 1997.

[7] Christiane Bracchi, Christophe Gefflot, and Frederic Paulin. Combining propagation information and search tree visualization using ilog opl studio. In *WLPE*, 2001.

[8] M. Carro and M. V. Hermenegildo. Tools for search tree visualization: the apt tool. In P. Deransart, M. V. Hermenegildo, and J. Malusynsky, editors, *Analysis and Visualization tools for constraint programming, constraint debugging, LNCS*, volume 1870, 2000.

[9] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[10] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: a generic graphical user interface for constraint logic programming. *Journal of Constraints, Special Issue on User-Interaction in Constraint Satisfaction*, 9(4):241–262, October 2004.

[11] C. Quigley J. Meng and L. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[12] J.Meng and L.Paulson. Experiments on supporting interactive proof using resolution. In *David Basin and Michael Rusinowitch (editors), IJCAR*, 2004.

[13] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[14] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[15] César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[16] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supŕieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[17] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Communications*, 15(2-3):91–110, 2002.

[18] Claudio Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5.

[19] Claudio Sacerdoti Coen and Enrico Tassi. Working with mathematical structures in type theory. In *TYPES*, 2007. To appear.

[20] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 1997. The MIT Press: Cambridge, MA, USA.

[21] H. Simonis and A. Aggoun. Search-tree visualization. In P. Deransart, M. V. Hermenegildo, and J. Malusynsky, editors, *Analysis and Visualization tools for constraint programming, constraint debugging, LNCS*, volume 1870, 2000.