Formal proof mining, a structure-oriented approach

Matthias Puech*

October 9, 2009

Large corpora of formal proofs have been developed over the years, that sit in the repositories of the various existing proof assistants. We advocate that these databases could be analysed by means of data-mining techniques, to help both the improvement of these tools on their usage, and the proof automation they provide. We suggest a technique for the analysis, *frequent subtree mining*, review the popular algorithms and suggest some adaptation to their specifications towards our application. We then formalize the theoretical framework of proof mining and review different issues we will have to deal with, along with some solutions.

Contents

Introduction

Context Formal proof is the field of computer science, logics and mathematics dealing with the design of computer tools and logical frameworks, helping a mathematician to express proofs in a formal language that can be automatically checked for validity by the computer. Some of these softwares – mathematical assistants – have reached a high degree of maturation and gained wide success: Coq [BBC+08],

*University of Bologna – University Paris 7

Isabelle, PVS, and the more recent one developed at the university of Bologna in the team of Andrea Asperti : Matita [ACTZ07]. All these tools are implementations of a logical framework with a convenient interactive language for writing proofs.

Over the years, large developments of formally verified proofs have been made in various fields of science, from mathematics to hardware verification, with the help of various iteractive proof assistants. These large proofs along with many others are stored in repositories and can be rechecked at will. Yet, the informations provided by these developments remains largely unexploited. Most proof assistants provide a very minimal set of interaction with previous developments : loading, reusing a result, searching by statement, and usually some kind of statement-based automation assimilated to proof-search (e.g. [AT09], [Pau99]). In these interactions, the proofs themselves are totally passive, kept only to testify of the validity of the reasoning. Unlike their equivalents in programming languages—libraries—proof assistants repositories do not only consist of names of objects (functions, classes etc.) but they also form a very structured and valuable source of information, both on the proof process and on the use one makes of the logical framework.

Empirical analysis By analysing these large databases of already human-formalized proofs with the help of *empirical methods* (statistics, data-mining, information theory, machine-learning), one could extract some useful informations.

The information extracted is expected to be of interest for a number of tasks. By analysing them "by hand", some common reasoning schemes could be isolated and serve as a basis for improving or designing high-level proof languages and procedures, validating their design *by the usage*. It could also, as a side-effect, be treated automatically or semi-automatically, to provide a valuable help to the user in a given situation about what has been previously done in similar situations – taking the form of suggestions or hints as in [MBDA06], or even to fully automatize some easy parts of proofs in a machine-learning fashion. The analysis of proof patterns proofs could help also to automatically factorize large proofs, to infer useful intermediate lemmas or to compress the representation of a proof.

This methodology has already been advocated and experimented in [DBL$^+$04] and [USPV08] with the help of standard data-mining concepts, leading to some interesting results. We propose a novel approach to the problem, by considering the whole structure of theproofs as our object of study, and not only some chosen features of them.

Representation of proofs According to the axiomatic method tradition, a formal proof, expressed in a given logical framework (first-order natural deduction, set theory, type theory...) is a *labelled tree with fixed arity*. The leafs are labelled by *axioms*, and internal nodes by *inference rules*. Each inference rule has a fixed number of sons, called its *arity*.

This tree representation is strictly isomorphic to a *term algebra*, and each proof-tree can be viewed as a term. This is the basis of an important observation in proof theory (the Curry-Howard isomorphism) as well as the theoretical basis of our investigation.

Methodology We postulate that among all informations contained in proof trees, the order of the chaining of rules, i.e. the very *structure* of the proof, is already of

crucial interest for our goal. By extracting from a big database of proof trees the common, recurring or generally interesting chaining of rules, that is some sub-trees of our trees, we wish to isolate some informations about the proof process, even some useful methods for future proofs.

We propose to data-mine proofs by means of *frequent subtree mining*. Frequent subtree mining is a relatively new research field, dedicated to the extraction of recurring patterns inside the structure of trees.

These methods will have to be adapted for our purpose : proof-terms are labeled, ordered trees with fixed arities and some other constraints, whereas most algorithms in the litterature have been devised for general trees. The direct reuse of a known algorithm is then doomed to be unadapted : the large majority of results would be non-relevant and the interesting ones would be drowned in this mass. Also, some of the notions used in the algorithms will have to be redefined : the notion of subtree differs in presence of terms, and we will have to refine the definition of frequency.

1 Frequent subtree mining, introduction and proposals

Frequent subtree mining is the research field dedicated to the exploration and data-mining of tree structures, by means of analysis of frequent patterns inside trees. It takes inspiration from the mature field of frequent itemset mining (finding frequent associations in sets of data) to devise efficient algorithms looking for frequent subtrees in sets of trees. See [CNMK05] for a pretty recent overview on the subject.

As an evolution of itemset mining, tree mining explores a more structured data structures, which is a common direction in the data-mining community. It is a particular case of graph mining, which is used for analysing large datasets organized in graph, such as network topology. By the ubiquous nature of trees in computer science, numerous applications emerged from this theoretical analysis, from Web to XML documents. We introduce the concepts at stake to then formalize the problem, give an example of a popular algorithm and sketch the design of a new, enhanced one, that would fit our needs for proof mining.

1.1 Preliminaries

Definition 1. *A* tree *is an undirected, connected acyclic graph $T = (V, E)$. It is* rooted *if we distinguish one vertex with no entering edge (the* root*). It is* ordered *if there is an order on each set of siblings. Given an alphabet Σ, it is* labeled *if there is a mapping $L : V \mapsto \Sigma$.*

Definition 2. *A* subtree $T' = (V', E')$ *of a tree $T = (V, E)$ is*

- induced *if $V' \subseteq V, E' \subseteq E$*
- bottom-up *if $V' \subseteq V, E' \subseteq E$ and $\forall v \in V', \forall (v, v') \in E, v' \in V'$*
- embedded *if $V' \subseteq V$, and $(v_1, v_2) \in E'$ only if v_1 is an ancestor of v_2.*

Additionaly, if T is ordered (resp. labeled), then T' must preserve the ordering (resp. the labeling) of T.

Lemma 1. *All three notions of subtrees are partial orders.*

Proof. (reflexivity) : Trivial. (transitivity) : By transitivity of $\subseteq$ for the induced and bottom-up case, by transitivity of the ancestor relation for the embedded case. (antisymmetry) : By antisymmetry of $\preceq$ and ancestor. □

Definition 3. *Given a partial order $\preceq$ on trees, A* distance $d_{\preceq}(T, U)$ *between trees T and U is an anti-monotone function from two trees to $\mathbb{R}$, i.e. $P \preceq P' \rightarrow \forall T, d(P', T) \geq d(P, T)$. The* frequency *of P in a set of trees $\mathcal{D}$ is* $\text{freq}_d(P, \mathcal{D}) = \sum_{T \in S} d(P, T)$.

1.2 The problem

We now define the specification of the frequent subtrees mining problem. For the sake of simplicity, and to restrict ourselves to the case serving our final goal, we focus on *rooted, ordered, labeled trees*.

Most algorithms in the litterature are based on the following definition of the problem. Unfortunately, it seems to have some inconvenients in practice. We first give the original formulation, and then some variants which realizations will be discussed later.

Frequent subtree mining Given :

- A subtree partial order $\preceq$
- A distance function $d_{\preceq}$
- A finite set of data trees $\mathcal{D}$
- a minimum frequency $m \in \mathbb{R}$

the problem of frequent subtree mining is to find the set $\mathcal{P}$ of trees s.t.

$$\forall P \in \mathcal{P}, \text{freq}_{d_{\preceq}}(P, \mathcal{D}) \geq m$$

One issue with this presentation is that the minimum frequency is a rather artificial and unintuitive parameter : there is no easy way to predict a good value for it, and, when using such an algorithm, we often have to relaunch the computation several times with different values of m. A generalization of this problem is to return the results in the order of decreasing frequency, eliminating purely the parameter m (it becomes a streaming algorithm).

Most frequent subtree mining We now try to find an ordered set $\mathcal{P}_{\leq}$, monotone wrt. $\leq$, i.e.

$$\forall P_1, P_2 \in \mathcal{P}_{\leq}, P_1 \leq P_2 \rightarrow \text{freq}_{d_{\preceq}}(P_1, \mathcal{D}) \leq \text{freq}_{d_{\preceq}}(P_2, \mathcal{D})$$

Unfortunately, the results returned by these (for the moment potential) algorithms would not correspond to the intended meaning of frequent subtrees. In fact, most of them would be considered as noise : consider one awaited result P, all of its subtrees $P' \preceq P$ will also be returned. Introducing an additional parameter to the algorithm, we can eliminate these redundancy :

Definition 4. *A* maximality function *wrt. a distance is a function M_d from a tree T to $\mathbb{R}$, free with respect to d (i.e. depending only on d).*

This function is used to give a weight to this redundancy : if $M_d(T)$ is high, it means that all its super-trees are probably much less interesting than T. We thus increase its rank among the solutions.

Maximally most frequent subtree mining is the extension of the *most frequent subtree mining* problem, weighted by a maximality function :

$$\forall P_1, P_2 \in \mathcal{P}_{\leq}, P_1 \leq P_2 \rightarrow \text{freq}_{d_{\preceq}}(P_1, \mathcal{D}) \times M_{d_{\preceq}}(P_1) \leq \text{freq}_{d_{\preceq}}(P_2, \mathcal{D}) \times M_{d_{\preceq}}(P_2)$$

1.3 A general but naive algorithm for FSM

It is easy to construct a generate-and-test algorithm for the first basic problem – *frequent subtree mining* – general enough to cover all notions of subtrees, distance and maximality. Given that :

- the set L of labels is finite,
- the freq_d and M function are computable

we can *enumerate* all possible trees T, and each time *test* for membership in the result set – i.e. test if $\text{freq}_d(T) \times M(T) \leq m$. The data set being finite, we know that the size of a subtree is bound by the size of the largest tree in the data set. Therefore, the algorithm terminates, and is sound and complete by construction (program 1).

```
fun freq_sub_mine (fun freq, set data, int m) {
  tree t = empty_tree;
  set r = empty_set;
  do {
    foreach d in data
      if (freq(t) >= m)
        r.add t;
    t.next_tree();
  } while (t != null)
  return r;
}
```

Program 1: The generate-and-test algorithm

This algorithm obviously suffers from its complexity : the frequency computation is repeated $|\mathcal{D}|$ times, and the enumeration of all trees with size smaller than the smallest $|D| \in \mathcal{D}$... we don't even want to know.

Some remarks will help us construct a realistic algorithm, by refining the generate-and-test approach :

- First, a crucial source of inefficency could be the enumeration of trees. It must not generate twice the same tree, but generate them all. One observation to simplify the process is the fact that subtrees of a tree form a (finite) lattice wrt. the subtree order.
- Secondly, an equivalent to the *a priori* criterion for intemset mining applies to trees : if a tree is not frequent, then all its super-trees have no chance of being frequent. Then, when encountering a non-frequent tree, we should skip all its super-trees in the enumeration (cut the branch in the enumeration lattice).

- Thirdly, blind generation of all possible labels during the enumeration is largely underoptimal, if not totally untractable. One approach is to restrict ourselves to only the labels present in the data set.

1.4 The FREQT algorithm

We will now focus on a popular algorithm for frequent subtree mining, devised in [AAK+02], solving the problem for *induced subtrees*.

The algorithm works in one preprocessing phase, followed by a main loop resembling closely the generate-and-test algorithm. The main improvement of FREQT over the naive algorithm is the enumeration procedure which, taking the form of a tree, allows to largely eliminate duplicate work. Besides, to determine the frequency of trees, FREQT uses an occurence list based approach.

In the preprocessing phase, we determine all frequent labels $\mathcal{L}$ of $\mathcal{D}$. Then, the enumeration tree is computed as follows : one extends a tree P_k in P_{k+1} by connecting a new node to the *rightmost path* (the path to the last node in the pre-order traversal). This new node will be labeled by a label in $\mathcal{L}$, and each possible extension of a tree P_k will be a son of it in the enumeration tree. This method has four advantages :

- It is complete (each possible tree will be generated),
- Each tree will be generated once,
- Successors can be determined efficiently (by only keeping a pointer on the rightmost node)
- It has the property that given a P_i, for all its sons $P_k, k \geq i, P_i \preceq P_k$. Therefore, if P_i is found to be non-frequent, we don't need to continue the enumeration of its sons, because they won't be frequent.

To determine the frequency of a tree P_k (each time we enumerate a new candidate), we keep a list of occurence of its rightmost node in $\mathcal{D}$. When extending it as P_{k+1}, we just have to count the occurence of the added node in each occurence of the database.

We iterate the main loop until there is no more branches in the enumeration tree to explore. Then all elements of the resulting tree are solutions.

1.5 Towards an algorithm for MMFSM

FREQT is already an efficient algorithm for frequent induced subtree mining, that is implemented and used in practice. However, some weaknesses made us look for a more versatile algorithm :

- The whole result depends on the m parameter, and it is difficult to foresee what value to give it, depending on the input. In a nutshell, it solves the FSM problem, not the MFSM. It is still possible to give m a very low value and sort the results afterwards, but in most real-life case, the first process is then too long to even terminate in a reasonable time. A more adequate parameter would be the number N of results expected.
- With no notion of maximality, most of the time the results are saturated by entries that are difficult to analyze : we have to check if each result is an interesting subtree, or if it appears in an other also interesting super-subtree.

This algorithm has also some downsides as far as the efficiency is concerned :

- One caracteristic of FREQT is not algorithmically satisfactory : the generation of the enumeration tree, as we saw, is guided by the set of frequent labels contained in $\mathcal{D}$. It means that during the enumeration, some extensions will be generated that aren't event present in the database (a frequent tree concatenated with a frequent node isn't necessarily even present). It would be easier and more efficient to directly generate the enumeration tree guided by the database content.
- Besides, some redundancy are still generated in the enumeration tree : indeed, all sons of P_k are subtrees, but some subtrees of P_k are not sons of it (for example, B is a subtree of $A(B)$, but not in the lineage of it). by generating the complete DAG of the subtree relation, we would allow to cut down more branches and spare some frequency computation.

Here are some guideance for the elaboration of an algorithm solving these issues : We drop the generation-test model, and proceed by iteration of a process composed of two phases : computation of the next generation of candidates, sorting of the results. We start with the empty tree ϵ, which is the parent of every node, and annotate it with all its occurences in the database (i.e. *all* nodes). We then compute the set of all its sons in the tree, and sort the resulting trees by frequency. The fact that there is much less different super-trees than labels makes the set significantly drop. And we continue with the sorted set of trees :

- Add all sons, annotate the occurences,
- Sort the results by frequency / keep N representant.

We iterate this process until the set of candidates contains only one tree : the whole database. At each step, we can check maximality of each tree, and order according to it.

The two algorithmical issues are made up : the test *is* the generation, so we don't generate any unuseful trees that will be dropped afterwards. There is also no need for pretreatment, since the first iteration of the process replaces it. Moreover, no redundancy is computed since the sorting phase takes over the test charge. The supplementary requirements are also met : there is no notion of minimum frequency, and maximality is easily taken into account.

2 Proof mining, requirements and directions

As introduced earlier, we propose to search interesting proof patterns, i.e. recurring, important or general pieces of proof. They are supposed to form common witnesses of the "know-how" of the mathematician. We believe that the tree-view of a proof already allows to see the proof process in a timely fashion : the focus is on the chaining of rules, not the static analysis of only one rule application. We will devise equivalent, or refined versions of the general "frequent subtree mining" techniques seen earlier, adapted to proofs. The first challenge of using these methods will be to carefully define the concepts :

- How to represent a proof (2.1),
- What forms a pattern in a proof (2.2),

- What makes a pattern interesting (e.g. distance, number of uses..., 2.3)

2.1 Preliminaries

Terms, substitutions, filtering We first define some basic concepts of rewriting theory:

Definition 5. *A* signature $\Sigma = (C, f_a)$ *is a set of named* constructors C*, along with an* arity function $f_a : C \to \mathbb{N}$.

Definition 6. *A* term *on a given signature* Σ *is a finite tree labeled with* C*, such that each node labeled with* $c_i \in C$ *has exactly* $f_a(c_i)$ *sons. We write* $\mathcal{T}(\Sigma)$ *to denote the set of terms engendered by a signature* Σ.

Definition 7. *Given an infinite set of variables* $\mathcal{X}$*, an* open term *or* pattern *on a signature* $\Sigma = (C, f_a)$ *is a term on the signature* $\Sigma_{\mathcal{X}} = (C \cup \mathcal{X}, f_a \uplus \{x \in \mathcal{X} \mapsto 0\})$. *We write* $\mathcal{T}(\Sigma, \mathcal{X})$ *to denote the set of patterns on* Σ *and* $\mathcal{X}$*. We write* $\mathrm{Vars}(p)$ *the set of variables in a pattern* p*. A pattern is* ground *if it contains no variables.*

Definition 8. *A* substitution σ *is a finite mapping* $\mathcal{V} \mapsto \mathcal{T}(\Sigma, \mathcal{X})$*. It is* ground *if it is* $\mathcal{V} \mapsto \mathcal{T}(\Sigma)$*. It is* grounding *wrt. a pattern* p *if* $p\sigma$ *is ground.*

Definition 9. *The* application $p\sigma$ *of a substitution (resp. ground substitution)* σ *to a pattern* p *is the replacement of each variables* X *of* p *by the term (resp. pattern)* $\sigma(X)$.

Definition 10. *Given* $p_1, p_2 \in \mathcal{T}(\Sigma, \mathcal{X})$ *and* $t \in \mathcal{T}(\Sigma)$*, we say that* p_1 filters p_2*, written* $p_1 \dot{\geq} p_2$*, if there exists a substitution* σ *s.t.* $p_1\sigma = p_2$*. Then* p_2 *is an* instance *of* p_1.

Formal system, derivation We now introduce some formally defined concepts about the usual notion of derivation.

Definition 11. *An* inference rule R *on a signature* Σ *is the pair of a pattern* $C \in \mathcal{T}(\Sigma, \mathcal{X})$ *(the* conclusion*) and a finite set of patterns* $P \subseteq T(\Sigma, \mathcal{X})$ *(the* premisses*). It is* effective *if* $\mathrm{Vars}(C) \subseteq \bigcup_{P_i \in P} \mathrm{Vars}(P_i)$*. We call* $f_a(R) = |P|$ *the* arity *of* R.

Definition 12. *An* annotated inference rule $R_{\mathcal{V}}$ *is an inference rule together with a name* R *and a set of variables* $\mathcal{V}$*. It is* effective *if* $\mathrm{Vars}(C) \subseteq \bigcup_{P_i \in P} \mathrm{Vars}(P_i) \cup \mathcal{V}$*. It is* minimally annotated *if* $\mathcal{V} = \mathrm{Vars}(C) \setminus \mathrm{Vars}(P)$*. It is* maximally annotated *if* $\mathcal{V} = \mathrm{Vars}(C)$.

Minimally and maximally annotated rules are of course effective, and it is easy to see that we can build a minimally annotated, effective rule $R_{\mathcal{V}}$ from a non-effective non-annotated rule R: just annotate it with the set of missing variables $\mathcal{V} = \mathrm{Vars}(C) \setminus \mathrm{Vars}(P)$.

Definition 13. *A* formal system S *is a set of annotated inference rules built on the same singature* Σ.

Definition 14. *A* derivation *of a pattern* T *(the* statement*) in a system* S *on* Σ *(noted* $t : T$*) is a term* t *built on the signature* $\Sigma_S = (\{R \in S\} \cup \{\_;,;\epsilon\} \cup \Sigma, f_{a_R} \uplus f_{a_\Sigma} \uplus \{(\_ \mapsto 2)\})$*, defined inductively as:*

- *if* $R_{\mathcal{V}} \in S$ *of arity* n *and* $t_1, \ldots, t_n$ *are derivations of repectively* $T_1, \ldots, T_n$, *and if* $P_1 \dot{\succeq} T_1, \ldots, P_n \dot{\succeq} T_n$ *with the same substitution* σ, *then the term* $R_{\mathcal{V}\sigma}(t_1, \ldots, t_2)$ *is a derivation of* $C\sigma$.

(only one rule, the base case being $n = 0$*).*

Note : The symbol _ denotes the subscript operation (like in LaTeX). It is used to annotate rules with terms.

Definition 15. *A formal system* S *on* Σ *is* effective *if there exists an algorithm which takes as input a term* $t \in \Sigma_S$ *and decides if* t *is a derivation of a pattern* $T \in \Sigma$ *in* S.

Theorem 1. *A formal system composed of effective rules is effective.*

Proof. (sketch) We recontruct the derivation step-by-step from the root of the tree. All missing informations are completed by the annotations of the rules since they are effective. □

Note that the converse is not true : there are some effective system S that have un-effective rules. This criterion is only a reasonable overestimation of the effectiveness of a system.

If a formal system is a proof system, i.e. used to model reasoning, then $t : T$ is read as " t is the proof(-term) of the statement T ". Here is an example of such a system :

Example 1. *Minimal natural deduction is a proof system where terms are constructed from the signature* $\Sigma_{DN} = \{\rightarrow^2; \vdash^2; ,^2; \epsilon^0\}$ *with the set of variables* $\mathcal{X} = \{\Gamma, \Delta, A, B, C \ldots\}$ *. It consists of the four inference rules :*

$$\text{INTRO}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \qquad \text{ELIM}\ \frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \text{WEAK}\ \frac{\Gamma \vdash B}{\Gamma, A \vdash B} \qquad \text{INIT}\ \frac{}{\Gamma, A \vdash A}$$

with respectively arity one, two, one and zero (an axiom). Here, the rule INIT *and* WEAK *aren't effective rules* per se *because the variables* A *don't appear in the premisses, but (minimally) annotated as* INIT_A *and* WEAK_A *, they become effective. The other rules are effective, so we consider that they are just annotated by an empty set of variables.*

A proof of the statement $\vdash (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$ *can then be represented by the following term :*

$$\text{INTRO}(\text{INTRO}(\text{INTRO}(\text{ELIM}(\text{WEAK}_A(\text{INIT}_{B\rightarrow C}), \text{ELIM}(\text{WEAK}_A(\text{WEAK}_{B\rightarrow C}(\text{INIT}_{A\rightarrow B})), \text{INIT}_A)))))$$

This syntax is isomorphic to the more standard de Bruijn version of the simply-typed λ*-calculus (*λ *is* INTRO*,* $t\ u$ *is* ELIM*,* S *is* WEAK*,* 0 *is* INIT*) :* $\lambda^{A\rightarrow B}\lambda^{B\rightarrow C}\lambda^{A}\ 1\ (2\ 0)$*, which stands for* $\lambda x^{A\rightarrow B} y^{B\rightarrow C} z^{A}.\ y\ (x\ z)$*. In this syntax though, we annotate the rule* λ *(*ELIM*) instead of* INIT *and* WEAK*, but the resulting system is still effective.*

We thus have a canonical way of viewing all formal system derivations, or proofs, as terms in a term algebra. This allows us to focus on the mining of these terms, without having to deal with general trees.

Subterms, subpatterns We now construct equivalents to the notions of subtrees introduced in definition 2, for terms and patterns.

Definition 16. *A* position ω *in a term is a node in its tree-representation, identified by the path from the root to* ω*. We write* $t|_\omega$ *to denote the term formed by the subtree at the node* ω*, and* $t[u]_\omega$ *the replacement of the subterm at* ω *by* u *in* t*.*

Definition 17. *We say that* t' *is a* subterm *of* t *if there exists an* ω *s.t.* $t|_\omega = t'$*. We call the* occurence number *of* t' *in* t *the number of different* ω*.*

Definition 18. *We say that* p *is a* subpattern *of* t *if there exists* ω *s.t.* $p \dot{\succeq} t|_\omega$*. We call the* occurence number *of* p *in* t *the number of different* ω*.*

Definition 19. *Given a signature* $\Sigma = (C, f_a)$ *and an infinite set of variables* $\mathcal{X}$*, an* embedded pattern *is a term on the signature* $\Sigma^E_{\mathcal{X}} = (C \cup \{\mathtt{X}[\mathtt{X_1} = \_; \ldots; \mathtt{X_n} = \_]\}, f_a \uplus \{\mathtt{X}[\mathtt{X_1} = \_ \ldots \mathtt{X_n} = \_] \mapsto n\}))$ *for all* n *and variables* $X, X_1 \ldots X_n$*.*

The notation gets heavy here : we are just adding a new n-ary construction to the terms, labeled with $n+1$ variables. The intuitive idea is to formalize the subterm properties in a modular way : $X[X_1 = t_1, \ldots, X_n = t_n] \preceq u$ stands for "u is a term which contains n subterms of the form $t_1 \ldots t_n$".

We now define the notions of filtering and substitution for embedded patterns mutually recursively:

Definition 20. *An embedded pattern* p filters *a term* t*, written* $p \dot{\succeq} t$ *if there exists a substitution* σ *s.t.* $p\sigma = t$*. The (new)* substitution $p\sigma$ *of embedded patterns is defined as follows :*

$$\begin{aligned}
(f(t_1, \ldots, t_n))\sigma &= f(t_1\sigma, \ldots, t_n\sigma) \\
(X[X_1 = t_1, \ldots, X_n = t_n])\sigma &= (X\sigma)\sigma \\
&\quad \text{(with } n \geq 1 \text{ and } Y[Y_1 = X_1, \ldots, Y_n = X_n] \dot{\succeq} X\sigma) \\
(X[])\sigma &= \sigma(X)
\end{aligned}$$

Definition 21. *An* embedded subpattern *is an embedded pattern whose head constructor is* $\mathtt{X}[\mathtt{X_1} = \_, \ldots, \mathtt{X_n} = \_]$

We remark that this new notion of pattern is more general than the previous one : a pattern is an embedded pattern whose variable nodes are all 0-ary; a subterm is an embedded subpattern with no variable nodes except the topmost one.

We now have equivalents for the three notions of subtrees for term algebras : A subterm is a bottom-up subtree. A subpattern with all variable leafs removed is an induced subtree. An embedded subpattern with all variable nodes ($\mathtt{X}[\mathtt{X_1} = \mathtt{t_1} \ldots \mathtt{X_n} = \mathtt{t_n}]$) removed except the topmost one is an embedded subtree.

2.2 Context-based automation

Although the extraction of frequent proof patterns could be interesting in itself to spot e.g. possible factorizations in proofs or general proof methods, it doesn't take into account the *context* in which they appear. Most proof assistants work by means of interaction with the user to gradually construct a proof(-term), presenting at each

step a *goal* to solve, and asking the user to enter a *tactic*, or a refinment making the proof progress. We would like to infer what the user proposed frequently *in a given context*, or *for a given kind of goal*.

In the framework of a formal system, this process of successive refinement boils down to the stepwise construction of a proof-pattern, each time presenting the statement corresponding to the variables of the term and asking for valid subpatterns to put in place of these variables, eventually reaching a ground and valid proof-term, i.e. a proof of the original statement.

Therefore, we can assimilate the *context* of a given subproof to the statement corresponding to its root. When annotated sufficiently, the proof terms contains enough information to let one reconstruct the statement by instantiation, and by data-mining at the same time the proof tree and the annotations (which are subterms), we expect the frequent patterns resulting from the analysis to contain informations about the annotations, i.e. about the context.

2.3 Distance, maximality of proof patterns

Let us now discuss the possible heuristics, parameters to our (hypothetical) algorithm. We saw that the notions of *distance* of a pattern—the measure of its interesting character—and of *maximality*—the measure of how redundant a pattern could be compared to its super-patterns—are both of crucial interest in order to retain only results of interest. We restrict ourself to the (non-embedded) patterns case (i.e. induced subtree mining), and present various possible version of each, from the simplest to the more elaborate.

Distance is, in the context of a term algebra, a function $d : \text{term} \times \text{pattern} \to \mathbb{R}$ such that for all p_1, p_2 if p_1 is a subpattern of p_2, then for all t, $d(t, p_1) \geq d(t, p_2)$

Definition 22. *The* occurence distance *between a term and a pattern* $d_{occ}(t, p)$ *is the occurence number of* p *in* t*.*

Definition 23. *The* complexity/occurence distance *is* $d_{cpx}(t, p) = d_{occ}(t, p) + \alpha|\text{Vars}(p)| + \beta\text{size}(p) + \gamma\text{size}(t)$ *where* α, β, γ *are weight constants.*

The idea here is that we may want to privilege bigger patterns with less variables (those that finish proofs faster), taking into account the size of the data term t.

Definition 24. *The* contextual distance *with respect to a formal system* S *is the function :*

$$d_{ctx}(t, p) = \frac{d_{occ}(t, p)}{|\text{Applicable}_S(p, t)|}$$

where :

- t *is a derivation of* T *in* S*,*
- $\text{Applicable}_S(p, t)$ *is the set of positions* ω *s.t.* $t[p]_\omega$ *(the replacement in* t *of the subterm* $t|_\omega$ *by the pattern* p*) is a derivation of* T *in* S*.*

This last notion of definition, although surely costly in terms of computation, seems much powerful. It is justified as follows : there may be frequent patterns of poor interest because they can be applied in a lot of situations, so they represent

a poor information. By relating the bare occurence count to the possible other uses a pattern could have, we take into account in some sense the value of the rare information.

Maximality is a function $M_d : \text{pattern} \to \mathcal{R}$ depending only in d.

Definition 25. *The* closedness *maximality criterion* $M_d^C = 1$ *if for all direct super-patterns* p' *of* p, $d(t, p') \neq d(t, p)$, *and* 0 *otherwise.*

Definition 26. *The* derivative maximality *is the function*

$$M_d^D = \sum_{p' \in SP(p)} d(t, p') - d(t, p)$$

where $SP(p)$ *is the set of all super-patterns of* p

The idea behind this last proposition is to weight the choice of a pattern by its relevance wrt. all its successors, i.e. the difference of distances.

2.4 Proof reduction and analysis

Mathematical proofs viewed as trees differ from usual trees in an important way : whereas a tree is a statical object, mathematical proofs form classes of equivalence according to some relations of computation. For example, cut-elimination, or β-reduction is the relation between two equivalent proofs, one involving the "factorization" of some reasoning steps and the other not (see [GLT89] for an introduction). Proof assistants usually also include a mechanism to define and reuse lemmas. The relation of computation between a proof involving a lemma and the one where its call is replaced directly by its lemma is usually called δ-reduction. Formalizations of mathematics involve a lot of these factorizations, and reduction is not meant to be performed : not only wouldn't one be interested in a *normal* proof, but it is often too large to even fit in a computer's memory. Therefore, we have to be able to mine these proofs *modulo* these classes of equivalence, but without computing their normal form.

Two challenges emerge from this fact for the analysis of patterns :

Analysis modulo computation Part of the sequentiality of a proof is *hidden* into its factorizations : the search of frequent pattern could be fooled by the use of a lemma. For example, an interesting pattern may be formed by part of the main proof (involving a lemma l) and the beginning of the proof of l. If we consider lemmas as opaque values or beta-redices statically, we may miss these interesting patterns.

Analysis of computation patterns At the same time, the way a proof has been cut or factorized by the mathematician provides some useful informations : it reflects the way a mathematician think about a given result. A witness of this fact is the way proofs are sketched in the litterature : often, only intermediate results are mentionned, the actual content being to be inferred by the reader. Statistically analysing where and how these cuts (or lemmas) occur in a large corpus of proofs

could provide interesting informations on the use of these *detours*, as well as a mean of automatizing this factorization. This could be done the same way we analyse proofs, by structuring their representation around the cuts.

None of the algorithms previously seen foresaw this kind of complex analysis, and it seems at first sight difficult to adapt them for these analysis. It would be a supplementary challenge to take into account these new requirements.

2.5 Practical issues and further works

Some preliminary experiments have been conducted to evaluate the feasability in terms of memory and time consumption. The existing implementation of the FREQT algorithm by Taku Kudo[1] has been used, and we have extracted the proof content of Coq's standard library in a sitable form (strict S-expressions) by directly translating the proof terms. The data file weighted about 12 megabytes and contained a 10.000 proof trees.

Sadly, but as expected, the issue of the analysis was not conclusive : the parameters of the algorithm were difficult to tweak, the computation time unpredictable and the results so overwhelming (some 10000 unsorted results), that the mining of useful informations was impossible. It was mainly due the the lack of maximality criterion to reduce the amount of sub-results, and the necessity to fix a quite arbitrary value for the minimum support.

Choice of a corpus For the analysis to be relevant, we need to directly start to work on large, real-life set of data. It means that we have to consider from the beginning the choice of a corpus to work on and probably cope with the complexity of the underlying language. Candidates could be for example the user-contributions of the proof assistants Coq[2], Matita[3] and Isabelle[4].

Representation of proofs Each corpus will be at first given in its own format(s). Depending on the needs, we will have to settle on a representation of proofs that is not too sparse (it should contain a maximum of informations), and not too explicit (for the patterns to remain interesting). For example, proofs in the Coq proof assistant are stored as terms in a well-defined algebra, but can also be read at the higher level of tactics – a tree of proof-state transformations. Both of them, along with the whole range of intermediate representations between them, could be chosen for the analysis. Also, it would be interesting to extract from the beginning the context annotations, so as to foresee the use of the result as a mean of automation.

Mining algorithm As we saw, existing algorithms do not fit our needs for this task. We need to devise and implement a new algorithm filling the requirements described in 1.5, as modularily as possible, to be able to experiment with the various parameters cited previously (concrete proof representation, matching (induced/embedded), criterion of maximality, frequency. . .).

[1] `http://chasen.org/ taku/software/freqt/`
[2] http://coq.inria.fr/distrib/current/contribs/
[3] http://matita.cs.unibo.it/library.shtml
[4] http://afp.sourceforge.net/

Extraction, interpretation and exploitation of the solutions Depending on the algorithm implemented and on the representation of proofs chosen, the found patterns will have certain properties, and certainly not all of them will be relevant, even with the right heuristics presented. Particularily, the criteria of maximality chosen will be critical to narrow down the candidates. For later automation, we should be able to draw relevant proof heuristics from these results, i.e. not to withdraw too much informations from them, to be able to recontruct a machine exploitable proof schema from the selected proof patterns.

References

[AAK+02] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proceedings of the Second SIAM International Conference on Data Mining*, pages 158–174, 2002.

[ACTZ07] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *J. Autom. Reasoning*, 39(2):109–139, 2007.

[AT09] A. Asperti and E. Tassi. An Interactive Driver for Goal-directed Proof Strategies. *Electronic Notes in Theoretical Computer Science*, 226:89–105, 2009.

[BBC+08] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual. *INRIA, version*, 8.1, 2008.

[CNMK05] Y. Chi, S. Nijssen, R.R. Muntz, and J.N. Kok. Frequent subtree mining – an overview. *Fundamenta Informaticæ*, 66(1):161–198, 2005.

[DBL+04] H. Duncan, A. Bundy, J. Levine, A. Storkey, and M. Pollet. The use of data-mining for the automatic formation of tactics. In *Workshop on Computer-Supported Mathematical Theory Development*. Citeseer, 2004.

[GLT89] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and types*. Cambridge University Press New York, 1989.

[MBDA06] A. Mercer, A. Bundy, H. Duncan, and D. Aspinall. PG Tips, a recommender system for an interactive prover. In *MathUI workshop*, 2006.

[Pau99] Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. Malarea sg1 - machine learner for automated reasoning with semantic guidance. In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 441–456, Berlin, Heidelberg, 2008. Springer-Verlag.