A

Proof-Pattern Search in Coq/SSReflect

Jónathan Heras, Department of Mathematics and Computer Science, University of La Rioja, Spain
Ekaterina Komendantskaya, School of Computing, University of Dundee, UK

ML4PG is an extension of the Proof General interface, allowing the user to invoke machine-learning algorithms and find proof similarities in Coq/SSReflect libraries. In this paper, we present four new improvements to ML4PG. First, a new method of "recurrent clustering" is introduced to collect statistical features from Coq terms. Now, the user can receive suggestions about similar definitions, types and lemma statements, in addition to proof strategies. Second, Coq proofs are split into patches to capture proof strategies that could arise at different stages of a proof. Third, we present a method to automatically generate Coq proofs based on ML4PG's output. Finally, we introduce several visualisation tools that help to interpret the families of similar theorems/terms generated by ML4PG.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic – Mechanical theorem proving; D.2.4 [**Software Engineering**]: Software/Program Verification – Formal methods.

Additional Key Words and Phrases: Coq/SSReflect, Proof Patterns, Recurrent Clustering, Pattern Recognition, Feature Extraction.

ACM Reference Format:
Jónathan Heras and Ekaterina Komendantskaya, 2014. Proof-Pattern Search in Coq/SSReflect. *ACM Trans. Comput. Logic* V, N, Article A (January YYYY), 24 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Development of Interactive Theorem Provers (ITPs) has led to the creation of big libraries and varied infrastructures for formal mathematical proofs. These frameworks usually involve thousands of definitions and theorems (for instance, there are approximately 4200 definitions and 15000 theorems in the formalisation of the Feit-Thompson theorem [Gonthier et al. 2013]). Parts of those libraries can often be re-applied in new domains; however, it is a challenge for expert and non-expert users alike to trace them, and find re-usable concepts and proof ideas.

The ML4PG ("Machine-Learning for Proof-General") tool [Komendantskaya et al. 2013; Heras and Komendantskaya 2013; Heras and Komendantskaya 2014b] was created to help the user in such a situation for the particular case of the Coq system [Coq development team 2013] and its SSReflect extension [Gonthier and Mahboubi 2010] developments. ML4PG uses statistical machine-learning algorithms to discover re-usable proof-patterns based on "previous experience" acquired from previous developments. A proof-pattern was defined as a correlation between the tactics and the types/shapes of subgoals resulting from the tactic applications within a few proof steps. The resulting tool could indeed find some interesting

The work was supported by EPSRC grants EP/J014222/1 and EP/K031864/1.
Author's addresses: Jónathan Heras and Ekaterina Komendantskaya, School of Computing, University of Dundee, UK

DOI:http://dx.doi.org/10.1145/0000000.0000000

— unexpected and yet relevant — proof-patterns, across different notation, types, and libraries. Our experiments spanned several subjects: basic mathematical infrastructures, Computer Algebra, Game Theory, and the certification of Java-like bytecode. The results are best summarised in [Heras and Komendantskaya 2013; Heras and Komendantskaya 2014b].

That initial approach had two inherent limitations. First, the essence of a Coq/SSReflect proof is not fully expressible by a sequence of applied tactics. The definitions, types, and shapes of auxiliary lemmas used in a proof can be much more sophisticated and conceptual than a proof script calling them. Therefore, although ML4PG could find interesting, and often useful, sequences of tactics; it could not go further to recognise, for instance, similar definitions. Second, the notion of a proof-pattern being "interesting" or useful is left to the user's judgement.

In this paper, we present the most recent extensions to ML4PG involving all kinds of Coq terms — type declarations, definitions and lemma statements — into the process of proof-pattern search. This required additional algorithms of feature extraction that reflect the mutual dependency of various proof objects in Coq's dependently-typed setting; see Sections 2 and 3. This major step in ML4PG development prompted other improvements. The initial ML4PG was considering features arising from the first 5 proof-steps in a proof, whereas now we treat every proof as a collection of proof patches, each potentially representing an interesting proof strategy. Moreover, if say 15th-20th step in one proof resembles a 115th-120th step in another, the tool is now able to detect such patterns deep down. The feature extraction algorithms for proof features have been further refined to include the data collected from Coq terms, and now the whole syntax of the chosen proof libraries is subject to *recurrent clustering* — a novel technique for ML4PG. All these extensions to proof-feature extraction are explained in Section 4.

ML4PG used to show, in response to the user's call, a set of similar proofs, with no hints of why these proofs are deemed similar. We now introduce two extensions that improve ML4PG's user-experience. First, we have designed a method to automatically proof theorems based on ML4PG clustering methods (cf. Section 5). Additionally, several visualisation tools (using automaton-shape or tree-shape representations) have been developed to show the proof-features that correlate, see Section 6. This partially addresses the drawback of the subjective approach to the pattern's "interestingness" — now the tool clearly declares correlation of which proof features defined the suggested proof pattern. Finally, in Section 7, we compare ML4PG with other machine-learning and searching approaches available in the literature, and conclude the paper in Section 8.

Examples we use throughout the paper come from several Coq and SSReflect libraries: the basic infrastructure of SSReflect [Gonthier and Mahboubi 2010], a matrix library [Garillot et al. 2009], a formalisation of persistent homology [Heras et al. 2013a], the HoTT library [Univalent Foundations Program 2013], two formalisation related to Nash equilibrium [Vestergaard 2006; Roux 2009], and the formalisation about Java-like bytecode presented in [Heras and Komendantskaya 2014b]. ML4PG is a part of standard Proof General distribution; the novel features we present here are available at [Heras and Komendantskaya 2014a].

2. FEATURE EXTRACTION FOR COQ TERMS

ML4PG uses (unsupervised) *clustering algorithms* [Bishop 2006] to find patterns in Coq syntax and proofs. Clustering algorithms divide data into n groups of similar objects (called *clusters*), where the value of n is a parameter provided by the user. In ML4PG, the value of n is automatically computed depending on the number of objects to cluster, and using the formula provided in [Komendantskaya et al. 2013; Heras et al. 2013b]. A detailed exposition of machine-learning algorithms involved in ML4PG can be found in [Komendantskaya

et al. 2013]. In this paper, our first goal consists in improving ML4PG's feature extraction algorithm.

Feature extraction [Bishop 2006] is a research area developing methods for discovery of statistically significant features in data. We adopt the following standard terminology. We assume there is a training data set, containing some samples (or objects). *Features* are given by a set of statistical parameters chosen to represent all objects in the given data set. If n features are chosen, one says that object classification is conducted in an n-dimensional space. For this reason, most pattern-recognition tools will require that the number of selected features is limited and fixed (sparse methods, like the ones applied in e.g. [Kaliszyk and Urban 2013; Kühlwein et al. 2013; Urban et al. 2008] are the exception to this rule). *Feature values* are rational numbers used to instantiate the features for every given object. If an object is characterised by n feature values, these n values together form a *feature vector* for this object. A function that assigns, to every object of the data set, a feature vector is called a *feature extraction function*. Normally, feature extraction is a data pre-processing stage, and it is separated from the actual pattern-recognition process.

Feature extraction from terms or *term trees* is common to most feature extraction algorithms implemented in theorem provers [Heras et al. 2013b; Kaliszyk and Urban 2013; Kühlwein et al. 2013; Urban et al. 2008]. In [Heras et al. 2013b], we introduced a feature extraction mechanism for ACL2 first-order terms. Here, that ACL2 method is substantially re-defined to capture the higher-order dependently-typed language of Coq.

The underlying formal language of Coq is known as the *Predicative Calculus of (Co)Inductive Constructions* (pCIC) [Coq development team 2013; Coquand and Huet 1988; Coquand and Paulin-Mohring 1990]. The terms of pCIC are built from the following rules:

Definition 2.1 (*pCIC term*).
– The sorts `Set`, `Prop`, `Type(i)` (`i`$\in \mathbb{N}$) are terms.
– The global names of the environment are terms.
– Variables are terms.
– If `x` is a variable and `T, U` are terms, then `forall x:T,U` is a term. If `x` does not occur in `U`, then `forall x:T,U` will be written as `T -> U`. A term of the form `forall x1:T1, forall x2:T2, ..., forall xn:Tn, U` will be written as `forall (x1:T1)(x2:T2)...(xn:Tn), U`.
– If `x` is a variable and `T, U` are terms, then `fun x:T => U` is a term. A term of the form `fun x1:T1 => fun x2:T2 => ... => fun xn:Tn => U` will be written as `fun (x1:T1)(x2:T2)...(xn:Tn)=> U`.
– If `T` and `U` are terms, then `(T U)` is a term – we use an uncurried notation (`(T U1 U2 ... Un)`) for nested applications (`(((T U1)U2)... Un)`).
– If `x` is a variable, and `T, U` are terms, then (`let x:=T in U`) is a term.

The syntax of Coq terms [Coq development team 2013] includes some terms that do not appear in Definition 2.1; e.g. given a variable `x`, and terms `T` and `U`, `fix name (x:T):= U` is a Coq term used to declare a recursive definition. The notion of a term in Coq covers a very general syntactic category in the Gallina specification language [Coq development team 2013] and corresponds to the intuitive notion of well-formed expression. However, for the purpose of concise exposition, we will restrict our notion of a term to Definition 2.1, giving the full treatment of the whole Coq syntax in the actual ML4PG implementation.

Definition 2.2 (*ML4PG term tree*). Given a Coq term `C`, we define its associated term tree as follows:
– If `C` is one of the sorts `Set`, `Prop` or `Type(i)`, then the term tree of `C` consists of one single node, labelled respectively by `Set:Type(0)`, `Prop:Type(0)` or `Type(i):Type(i+1)`.

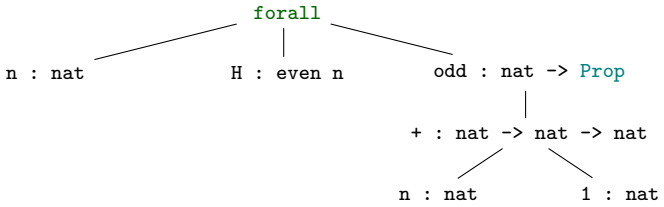

Fig. 1. *ML4PG term tree for the term* `forall (n : nat) (H : even n), odd (n + 1)`.

– If `C` is a name or a variable, then the term tree of `C` consists of one single node, labelled by the name or the variable itself together with its type.
– If `C` is a term of the form `forall (x1:T1)(x2:T2)...(xn:Tn), U` (analogously for `fun (x1:T1)(x2:T2)...(xn:Tn)=> U`); then, the term tree of `C` is the tree with the root node labelled by `forall` (respectively `fun`) and its immediate subtrees given by the trees representing `x1:T1`, `x2:T2`, `xn:Tn` and `U`.
– If `C` is a term of the form `let x:=T in U`, then the term tree of `C` is the tree with the root node labelled by `let`, having three subtrees given by the trees corresponding to `x`, `T` and `U`.
– If `C` is a term of the form `T -> U`, then the term tree of `C` is represented by the tree with the root node labelled by `->`, and its immediate subtrees given by the trees representing `T` and `U`.
– If `C` is a term of the form `(T U1 ... Un)`, then we have two cases. If `T` is a name, the term tree of `C` is represented by the tree with the root node labelled by `T` together with its type, and its immediate subtrees given by the trees representing `U1`,..., `Un`. If `T` is not a name, the term tree of `C` is the tree with the root node labelled by `@`, and its immediate subtrees given by the trees representing `T`, `U1,...,Un`.

Note that ML4PG term trees consist of two kinds of nodes: *Gallina* and *term-type* nodes. The Gallina nodes are labelled by Gallina-keywords and special tokens such as `forall`, `fun`, `let` or `->` (from now on, we will call them Gallina tokens); and the term-type nodes are labelled by expressions of the form `t1:t2` where `t1` is a sort, a variable or a name, and `t2` is the type of `t1`.

Example 2.3. Given the term `forall (n : nat)(H : even n), odd (n + 1)`, its ML4PG term tree is depicted in Figure 1.

We represent ML4PG term trees as feature matrices, further to be flattened into feature vectors for clustering. A variety of methods exists to represent trees as matrices, for instance using adjacency or incidence matrices. The adjacency matrix and the various previous methods of feature extraction (e.g. [Kaliszyk and Urban 2013; Kühlwein et al. 2013; Urban et al. 2008]) share the following common properties: different library symbols are represented by distinct features, and the feature values are binary. For big libraries and growing syntax, feature vectors grow very large (up to 10^6 in some experiments) and at the same time very sparse, which implies the use of sparse machine-learning methods.

We develop a new compact method that tracks a large (potentially unlimited) number of Coq terms by a finite number of features and an unlimited number of feature-values. In our method, the features are given by two properties common to all possible term trees: the term tree depth and the level index of nodes. The most important information about the term is then encoded by improving precision of feature values using rational-valued feature-extraction functions. Taking just 300 features, the new feature-extraction method recursively adjusts the feature values, adapting to the growing language syntax. The resulting feature vectors have an average density ratio of 60%.

Table I. *ML4PG term tree matrix for* `forall (n : nat) (H : even n), odd (n+1).`

	level index 0	level index 1	level index 2
td0	([forall]$_{Gallina}$,-1,-1)	(0,0,0)	(0,0,0)
td1	([n]$_{term}$,[nat]$_{type}$,0)	([H]$_{term}$,[even n]$_{type}$,0)	([odd]$_{term}$,[nat-> Prop]$_{type}$,0)
td2	([+]$_{term}$,[nat -> nat -> nat]$_{type}$,2)	(0,0,0)	(0,0,0)
td3	([n]$_{term}$,[nat]$_{type}$,0)	([1]$_{term}$,[nat]$_{type}$,0)	(0,0,0)

Given a Coq expression, we can differentiate its term and type components; the feature values capture information from these components and also the structure of the tree. In particular, each tree node is encoded by distinct feature values given by a triple of rational numbers to represent the term component, the type component, and the level index of the parent node in the term tree, cf. Table I. Our feature extraction method is formalised in the following definitions.

Definition 2.4 (*Term tree depth level and level index*). Given a term tree T, the *depth* of the node t in T, denoted by *depth(t)*, is defined as follows:
– $depth(t) = 0$, if t is a root node;
– $depth(t) = n + 1$, where n is the depth of the parent node of t.
The *nth level* of T is the ordered sequence of nodes of depth n — using the classical representation for trees, the order of the sequence is given by visiting the nodes of depth n from left to right. The *level index* of a node with depth n is the position of the node in the nth level of T. We denote by $T(i, j)$ the node of T with depth i and index level j.

We use the notation $M[\mathbb{Q}]_{n\times m}$ to denote the set of matrices of size $n \times m$ with rational coefficients.

Definition 2.5 (*ML4PG term tree feature extraction*). Given a term $\mathtt{t}$, its corresponding term tree $T_{\mathtt{t}}$, and three injective functions $[.]_{term} : Coq\ terms \to \mathbb{Q}^+$, $[.]_{type} : Coq\ terms \to \mathbb{Q}^+$ and $[.]_{Gallina} : Gallina\ tokens \to \mathbb{Q}^-$, then the feature extraction function $[.]_M = \langle [.]_{term}, [.]_{type}, [.]_{Gallina} \rangle : Coq\ terms \to M[\mathbb{Q}]_{10\times 10}$ builds the *term tree matrix of* $\mathtt{t}$, $[\mathtt{t}]_M$, where the (i, j)-th entry of $[\mathtt{t}]_M$ captures information from the node $T_{\mathtt{t}}(i, j)$ as follows:
– if $T_{\mathtt{t}}(i, j)$ is a Gallina node g, then the (i, j)th entry of $[\mathtt{t}]_M$ is a triple $([g]_{Gallina}, -1, p)$ where p is the level index of the parent of g.
– if $T_{\mathtt{t}}(i, j)$ is a term-type node `t1:t2`, then the (i, j)th entry of $[\mathtt{t}]_M$ is a triple $([t1]_{term}, [t2]_{type}, p)$ where p is the level index of the parent of the node.

In the above definition, we fix the maximum depth and maximum level index of a node to 10; this makes the feature extraction mechanism uniform across all Coq terms appearing in the libraries. We may lose some information if pruning is needed, but the chosen size works well for most terms appearing in Coq libraries. If a term tree does not fit into 10×10 term tree dimensions, its 10×10 subtree is still considered by ML4PG. The term tree matrix is flattened into a feature vector, and each triple will be split into three components of the vector, giving a feature vector size of 300, still smaller than in sparse approaches [Kaliszyk and Urban 2013; Kühlwein et al. 2013; Urban et al. 2008].

In Definition 2.5, we deliberately specify the functions $[.]_{Gallina}$, $[.]_{term}$ and $[.]_{type}$ just by their signature. The function $[.]_{Gallina}$ is a predefined function. The number of Gallina tokens (`forall`, `fun`, `->` and so on) is fixed and cannot be expanded by the Coq user. Therefore, we know in advance all the Gallina tokens that can appear in a development, and we can assign a concrete value to each of them. The function $[.]_{Gallina} : Gallina\ tokens \to \mathbb{Q}^-$ is an injective function carefully defined to assign close values to similar Gallina tokens and more distant numbers to unrelated tokens — see Appendix A for the exact encoding.

The functions $[.]_{term}$ and $[.]_{type}$ are dynamically re-defined for every library and every given proof stage, to adapt to the changing syntax. In practice, there will be new $[.]_{term}$ and $[.]_{type}$ functions computed whenever ML4PG is called. This brings the element of the

"acquired knowledge/experience" to the machine-learning cycle, as will be formalised in the next section.

3. RECURRENT TERM CLUSTERING

The previous section introduced a method of defining statistically significant features. It remains to define the functions $[.]_{term}$ and $[.]_{type}$ that will determine the feature values. These functions must be sensitive to the structure of terms, assigning close values to similar terms, and more distant values to unrelated terms.

A term `t` is represented by the 300 feature values of $[\texttt{t}]_M$. The values of $[.]_M$ for variables and pre-defined sorts in `t` are fixed, but the values of user-defined terms (and types!) contained in `t` have to be computed recursively, based on the structures of their definitions, and using clustering to compute their feature vectors, and their representative values for $[\texttt{t}]_M$. It is the nature of functional languages to have terms depending on other terms, and feature extraction/clustering cycle is repeated recursively to reflect complex mutual term dependencies as feature values. We call this method *recurrent clustering*: the function $[.]_M$ automatically (and recurrently) adapts to the given libraries and the current proof stage. This differs from the standard machine-learning approach (and the old version of ML4PG), where the process of feature extraction is separated from running pattern-recognition algorithms. Here, one is a crucial part of another.

When Coq objects are divided into clusters, a unique integer number is assigned to each cluster. Clustering algorithms compute a *proximity value* (ranging from 0 to 1) to every object in a cluster to indicate the certainty of the given example belonging to the cluster. The cluster numbers and the proximity values are used in the definitions of $[.]_{term}$ and $[.]_{type}$ below.

Definition 3.1. Given a term `t` of a Coq library, the functions $[.]_{term}$ and $[.]_{type}$ are defined respectively for the term component `t1` and the type component `t2` of every term-type node in the ML4PG term tree of `t` as follows:

- $[\texttt{t1}]_{term/type} = i$, if `t1` is the ith distinct variable in `t`.
- $[\texttt{t1}]_{term/type} = 100 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$, if `t1` is the ith element of the set $\{\texttt{Set}, \texttt{Prop}, \texttt{Type(0)}, \texttt{Type(1)}, \texttt{Type(2)}, \ldots\}$.
- $[\texttt{t1}]_{term} = 200 + 2 \times j + p$, where j is a number of a cluster C_j computed by the latest run of term clustering, such that p is the proximity value of `t1` in C_j.
- $[\texttt{t2}]_{type} = 200 + 2 \times j + p$, where j is a number of a cluster C_j computed by the latest run of type clustering (i.e. term clustering restricted to types), such that p is the proximity value of `t2` in C_j.

Note the recurrent nature of the functions $[.]_{term}$ and $[.]_{type}$ where numbering of components of `t` depends on the term definitions and types included in the library, assuming those values are computed by iterating the process back to the basic definitions. In addition, the function $[.]_{term}$ internally uses the function $[.]_{type}$ in the recurrent clustering process and *vice versa*.

In the above definition, the variable encoding reflects the number and order of unique variables appearing in the term, note its similarity to the de Bruijn indexes. In the formula for sorts, $\sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$ reflects the close relation among sorts, and 100 is used to distinguish sorts from variables and names. Finally, the formula $200 + 2 \times j + p$ assigns $[\texttt{t1}]$ (or $[\texttt{t2}]$) a value within $[200 + 2 \times j, 200 + 2 \times j + 1]$ depending on the statistical proximity of `t1` (or `t2`) in cluster j. Thus, elements of the same cluster have closer values comparing to the values assigned to elements of other clusters, sorts, and variables. The formula is the same for the functions $[.]_{term}$ and $[.]_{type}$, but it is computed with different clusters and the values occur in different cells of the term-tree matrices (cf. Definition 2.5); thus, clustering algorithms distinguish terms and types on the level of features rather than feature values.

We can now state the main property of the ML4PG feature extraction.

PROPOSITION 3.2. *Let $\mathcal{T}$ be the set of Coq terms whose trees have maximum depth* 10 *and level index* 10. *Then, the function $[.]_M$ restricted to $\mathcal{T}$ is a one-to-one function.*

Once the feature values of ML4PG term tree matrices have been computed, we can cluster these matrices and obtain groups of similar terms. In particular, ML4PG can be used to cluster definitions, types and lemma statements. We finish this section with some clusters discovered among the 457 definitions of the basic infrastructure of the SSReflect library [Gonthier and Mahboubi 2010].

Example 3.3. We include here 3 of the 91 clusters discovered by ML4PG automatically in the SSReflect library of 457 terms (across 12 standard files), within 5–10 seconds. Note that this example of cluster-search is not goal-oriented, ML4PG discovers patterns without any user guidance, and offers the user to consider term similarities of which he may not be aware.

— Cluster 1:

```
Fixpoint eqn (m n : nat) :=
  match m, n with
  | 0, 0 => true | m'.+1, n'.+1 => eqn m' n'
  | _, _ => false end.
Fixpoint eqseq (s1 s2 : seq T) :=
  match s1, s2 with
  | [::], [::] => true | x1 :: s1', x2 :: s2' => (x1 == x2) && eqseq s1' s2'
  | _, _ => false end.
```

— Cluster 2:

```
Fixpoint drop n s := match s, n with | _ :: s', n'.+1 => drop n' s' | _, _ => s end.
Fixpoint take n s := match s, n with | x :: s', n'.+1 => x :: take n' s' | _, _ => [::] end.
```

— Cluster 3:

```
Definition flatten := foldr cat (Nil T).
Definition sumn := foldr addn 0.
```

The first cluster contains the definitions of equality for natural numbers and lists — showing that ML4PG can spot similarities across libraries. The second cluster discovers the relation between `take` (takes the first n elements of a list) and `drop` (drops the first n elements of a list). The last pattern is less trivial of the three, as it depends on other definitions, like `foldr`, `cat` (concatenation of lists) and `addn` (sum of natural numbers). Recurrent term clustering handles such dependencies well: it assigns close values to `cat` and `addn`, since they have been discovered to belong to the same cluster. Note the precision of ML4PG clustering. Among 457 terms it considered, 15 used `foldr`, however, Cluster 3 contained only 2 definitions, excluding e.g. `Definition allpairs s t:=foldr (fun x => cat (map (f x)t)) [::] s` ; `Definition divisors n:=foldr add_divisors [:: 1] (prime_decomp n)` or `Definition Poly:=foldr cons_poly 0.` — this is due to the recurrent clustering process since functions like `add_divisors` or `cons_poly` are not clustered together with functions `cat` and `addn`.

To summarise, there are three main properties that distinguish ML4PG pattern search from standard Coq search commands:

— the user does not have to know and provide any search pattern;
— the discovered clusters do not have to follow a "pattern" in a strict sense (e.g. neither exact symbol names nor their order make a pattern), but ML4PG considers structures and background information found in the library; and,
— working with potentially huge sets of Coq objects, ML4PG makes its own intelligent discrimination of more significant and less significant patterns, as example with `foldr`

has shown. This is opposed to the classic search for `foldr` pattern that would present the user with a set of 15 definitions.

ML4PG can also work in a goal-directed mode, and discover only clusters of terms that are similar to the given term `t`. This can speed-up the proof development in two different ways. In addition, clustering will provide definitions of terms similar to `t`; hence, the proofs of the theorems involving those terms may follow similar patterns. Clustering can also discover that a newly defined term `t` was previously defined (perhaps in a different notation, as ML4PG works with structures across notations); in that case, the user can use the existing library definition and all its background theory instead of defining it from scratch.

4. RECURRENT PROOF CLUSTERING

The method presented in the previous section can cluster similar statements of all Coq terms, including definitions and theorems. However, this method does not capture the interactive nature of Coq proofs. In this section, we involve proofs into the recurrent clustering of Coq libraries.

In [Komendantskaya et al. 2013], we introduced a feature extraction method for Coq proofs capturing the user's interaction through the applied tactics. That method traced low-level properties present in proof's subgoals, e.g. "the top symbol" or "the argument type". Further, these features were taken in relation to the statistics of user actions on every subgoal: how many and what kind of tactics he applied, and what kind of arguments he provided to the tactics. Finally, a few proof-steps were taken in relation to each other. This method had two drawbacks.

(1) It was focused on the first five proof-steps of a proof; therefore, some information was lost. We address this issue by implementing automatic split of each proof into proof-patches; thus, allowing ML4PG to analyse a proof by the properties of the patches that constitute the proof.

(2) The method assigned most feature-values blindly; thus, being insensitive to many important parameters, such as e.g. the structure of lemmas and hypotheses used as tactic arguments within a proof. The previous section gave us the way of involving all Coq objects into recurrent feature re-evaluation.

Definition 4.1 (*Coq proof*). Given a statement S in Coq, a *Coq proof* of S is given by a sequence of triples $((\Gamma_i, G_i, T_i))_{0\leq i\leq n}$ where Γ_i is a context, G_i is a goal and T_i is a sequence of tactics satisfying:

- $G_0 = S$, and for all i, Γ_i is the context of the goal G_i,
- for all i with $0 < i \leq n$, Γ_i, G_i are respectively the context and goal obtained after applying T_{i-1}, and the application of T_n completes the proof.

In this paper, we focus on the goals and tactics of Coq proofs; thus, we do not consider the contexts and denote the Coq proof $((\Gamma_i, G_i, T_i))_{0\leq i\leq n}$ by $((G_i, T_i))_{0\leq i\leq n}$. Involving contexts into proof-pattern search may be a subject for future work.

Example 4.2. Table II shows the Coq proof of the following statement:

$$\forall g : \mathbb{N} \to \mathbb{Z} \implies \sum_{0\leq i\leq n} (g(i+1) - g(i)) = g(n+1) - g(0)$$

One small proof may potentially resemble a fragment of a bigger proof; also, various small "patches" of different big proofs may resemble.

Definition 4.3 (*Proof-patch*). Given a *Coq proof* $C = ((G_i, T_i))_{0\leq i\leq n}$, a *proof-patch* of C is a subsequence of at most 5 consecutive pairs of C.

From proof-patches, we can construct their feature matrices. We will shortly define the feature extraction function $[.]_P = \langle [.]_M, [.]_{tac} \rangle : \mathit{proof\ patches} \to M[\mathbb{Q}]_{5\times 6}$, where $[.]_{tac}$ is

Table II. *Proof for the lemma of Example 4.2 in SSReflect.*

Goals and Subgoals	Applied Tactics
G_0)$\forall\ n, \sum_{i=0}^{n}(g(i+1)-g(i)) = g(n+1)-g(0)$	T_0) `case : n => [|n _].`
G_1) $\sum_{i=0}^{0}(g(i+1)-g(i)) = g(1)-g(0)$	T_1) `by rewrite big_nat1.`
G_2) $\sum_{i=0}^{n+1}(g(i+1)-g(i)) = g(n+2)-g(0)$	T_2) `rewrite sumrB big_nat_recr big_nat_recl addrC addrC -subr_sub -!addrA addrA.`
G_3)$g(n+2)+\sum_{i=0}^{n} g(i+1)-\sum_{i=0}^{n} g(i+1)-g(0) = g(n+2)-g(0)$	T_3) `move : eq_refl.`
G_4) $\sum_{i=0}^{n} g(i+1) == \sum_{i=0}^{n} g(i+1) \rightarrow$ $g(n+2)+\sum_{i=0}^{n} g(i+1)-\sum_{i=0}^{n} g(i+1)-g(0) = g(n+2)-g(0)$	T_4) `rewrite -subr_eq0.`
G_5) $\sum_{i=0}^{n} g(i+1)-\sum_{i=0}^{n} g(i+1) == 0 \rightarrow$ $g(n+2)+\sum_{i=0}^{n} g(i+1)-\sum_{i=0}^{n} g(i+1)-g(0) = g(n+2)-g(0)$	T_5) `move/eqP => ->.`
G_6)$g(n+2)+0-g(0) = g(n+2)-g(0)$	T_6) `by rewrite sub0r.`
□	`Qed.`

an injective function that has been introduced to assign values to tactics. We have defined two versions of $[.]_{tac}$: one for Coq tactics and another for SSReflect tactics. In the SSReflect case, we divide the tactics into 7 groups and assign similar values to each tactic in the group, see Table III. Analogously for Coq tactics, cf. Appendix B.

Definition 4.4 (*Proof-patch matrix*). Given a Coq proof $C = ((G_i, T_i))_{0\leq i\leq n}$, and a proof patch $p = ((G_{i_0}, T_{i_0}), \ldots, (G_{i_4}, T_{i_4}))$ of C, the feature extraction function $[.]_P$: *proof patches* $\rightarrow M[\mathbb{Q}]_{5\times 6}$ constructs the *proof-patch matrix* $[p]_P$ as follows:

— the $(j,0)$-th entry of $[p]_P$ is a 4-tuple $([T^1_{i_j}]_{tac}, [T^2_{i_j}]_{tac}, [T^3_{i_j}]_{tac}, [T^r_{i_j}]_{tac})$ where $T^1_{i_j}, T^2_{i_j}$ and $T^3_{i_j}$ are the three first tactics of T_{i_j}, and $T^r_{i_j}$ is the list of the rest of tactics of T_{i_j},
— the $(j,1)$-th entry of $[p]_P$ is the number of tactics appearing in T_{i_j},
— the $(j,2)$-th entry of $[p]_P$ is a 4-tuple $([t_1]_{type}, [t_2]_{type}, [t_3]_{type}, [t_{i_j}]_{type})$ where t_1, t_2 and t_3 are the three first argument-types of T_{i_j}, and t_{i_j} is the set of the rest of the argument-types of T_{i_j} (insensitive to order or repetition),
— the $(j,3)$-th entry of $[p]_P$ is a 4-tuple $([l_{i_{j_1}}]_{term}, [l_{i_{j_2}}]_{term}, [l_{i_{j_3}}]_{term}, [l_{i_j}]_{term})$ where $l_{i_{j_1}}$, $l_{i_{j_2}}$ and $l_{i_{j_3}}$ are the three first lemmas applied in T_{i_j} and l_{i_j} is the list of the rest of lemmas used in T_{i_j} (sensitive to order and repetition),
— the $(j,4)$-th entry of $[p]_P$ is a triple $([s_1]_{term}, [s_2]_{term}, [s_3]_{term})$ where s_1, s_2 and s_3 are respectively the top, second, and third symbol of G_{i_j},
— the $(j,5)$-th entry of $[p]_P$ is the number of subgoals after applying T_{i_j} to G_{i_j}.

Example 4.5. Given the proof of Example 4.2 and the proof-patch $((G_i, T_i))_{0\leq i\leq 4}$, the top table of Table IV shows their proof-patch matrix.

The *proof-patch method* considers several proof-patches to collect information from a concrete proof. In particular, given a Coq proof $C = ((G_i, T_i))_{0\leq i\leq n}$, the proof C can be split into patches $C_0, \ldots, C_m$ where $m = \lceil \frac{n}{5} \rceil + 1$. The patches are defined as follows: $C_j = ((G_j, T_j), \ldots, (G_{j+4}, T_{j+4}))$ for $0 \leq j < m$ (some patches might contain less than 5 proof-steps), and $C_m = ((G_{n-4}, T_{n-4}), \ldots, (G_n, T_n))$ — the last patch captures the last five proof-steps.

Table III. *Formulas computing the value of SSReflect tactics.* they serve to assign closer values to the tactics within each of the seven groups, and more distant numbers across the groups. If a new tactic is defined, ML4PG automatically assigns a new number to it, using the next available natural number n in the formula $n + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.

∗ Bookkeeping (b = {`move:`, `move =>`}):	$[b_i]_{tac} = 1 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$ (where b_i is the ith element of b).
∗ Case and Induction (c = {`case`, `elim`}):	$[c_i]_{tac} = 2 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.
∗ Discharge (d = {`apply`, `exact`, `congr`}):	$[d_i]_{tac} = 3 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.
∗ Simplification (s = {`//`, `/=`, `//=`}):	$[s_i]_{tac} = 4 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.
∗ Rewrite:	$[\texttt{rewrite}]_{tac} = 5$.
∗ Forward Chaining (f = {`have`, `suff`, `wlog`}):	$[f_i]_{tac} = 6 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.
∗ Views and reflection (v = {`move/`, `apply/`, `elim/`, `case/`}):	$[v_i]_{tac} = 7 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.

Example 4.6. Using the proof-patch method, we can split the proof presented in Example 4.5 into three proof-patches $((G_i, T_i))_{0 \leq i \leq 4}$, $((G_5, T_5), (G_6, T_6))$ and $((G_i, T_i))_{2 \leq i \leq 6}$; the corresponding proof-patch matrices are given in Table IV.

The proof-patch method together with the feature function $[.]_P$ solve the two drawbacks of the old method [Komendantskaya et al. 2013]: the new method captures information about the whole proof and the feature values are dynamically computed to assign close values to similar terms, types, tactics and lemma statements used as tactic arguments.

We finish this section with a case study that illustrates the use of the proof-patch method, and shows the differences with the results obtained with the old method [Komendantskaya et al. 2013]. This case study concerns the discovery of proof patterns in mathematical proofs across formalisations of apparently disjoint mathematical theories: Linear Algebra, Combinatorics and Persistent Homology (across 758 lemmas and 5 libraries). In this scenario, we use statistically discovered proof patterns to advance the proof of a given "problematic" lemma. Namely, a few initial steps in its proof are clustered against several mathematical libraries. We deliberately take lemmas belonging to very different SSReflect libraries. The lemma introduced in Example 4.2 is a basic fact about summations. Lemma 4.7 states a result about *nilpotent* matrices (a square matrix M is *nilpotent* if there exists an n such that $M^n = 0$). Finally, Lemma 4.8 is a generalisation of the *fundamental lemma of Persistent Homology* [Heras et al. 2013a].

LEMMA 4.7. *Let M be a square matrix and n be a natural number such that $M^n = 0$, then $(1 - M) \times \sum_{i=0}^{n-1} M^i = 1$.*

LEMMA 4.8. *Let $\beta_n^{k,l} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$, then*

$$\sum_{0 \leq i \leq k} \sum_{l < j \leq m} (\beta_n^{i,j-1} - \beta_n^{i,j}) - (\beta_n^{i-1,j-1} - \beta_n^{i-1,j}) = \beta_n^{k,l} - \beta_n^{k,m}.$$

When proving Lemma 4.7, the user may call ML4PG after completing a few standard proof steps: apply induction and solve the base case using rewriting. At this point it is difficult, even for an expert user, to get the intuition that he could reuse the proofs from Example 4.2 and Lemma 4.8. There are several reasons for this. First of all, the formal proofs of these lemmas are in different libraries; then, it is difficult to establish a conceptual connection among them. Moreover, although the three lemmas involve summations, the types of the terms of those summations are different. Therefore, search based on types or keywords would not help. Even search of all the lemmas involving summations does not provide a clear suggestion, since there are more than 250 lemmas (a considerable number for handling them manually) — a clever search-pattern will considerably reduced the number of suggested lemmas, but such a pattern is not trivial at all.

However, if only the lemmas from Example 4.2 and Lemma 4.8 are suggested when proving Lemma 4.7, the user would be able to spot the following common proof pattern.

Table IV. *Proof-patch matrices for the proof of Example 4.5.* **Top.** Proof-patch matrix of the patch $((G_i, T_i))_{0 \leq i \leq 4}$. **Centre.** Proof-patch matrix of the patch $((G_i, T_i))_{5 \leq i \leq 6}$ (rows that are not included in the table are filled with zeroes). **Bottom.** Proof-patch matrix of the patch $((G_i, T_i))_{2 \leq i \leq 6}$. Where we use notation EL, ML4PG gathers the lemma names: (`addrC`, `addrC` , `subr_sub`, ...).

	tactics	*n*	*arg type*	*arg*	*symbols*	*goals*
g1	$([case]_{tac}, 0, 0, 0)$	1	$([nat]_{type}, 0, 0, 0)$	$([Hyp]_{term}, 0, 0, 0)$	$([\forall]_{term}, [=]_{term}, [sum]_{term})$	2
g2	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([big\_nat1]_{term}, 0, 0, 0)$	$([=]_{term}, [\sum]_{term}, [-]_{term})$	0
g3	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}$, $[Prop]_{type}$, $[Prop]_{type}$, $[Prop]_{type})$	$([surB]_{term}$, $[big\_nat\_recr]_{term}$ $[big\_nat\_recl]_{term}$ $[EL]_{term})$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g4	$([move :]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([eq\_refl]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g5	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([subr\_eq0]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1

	tactics	*n*	*arg type*	*arg*	*symbols*	*goals*
g1	$([move/]_{tac}, [->]_{tac}, 0, 0)$	2	$([Prop]_{type}, 0, 0, 0)$	$([eq\_refl]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g2	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([subr\_eq0]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1

	tactics	*n*	*arg type*	*arg*	*symbols*	*goals*
g1	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}$, $[Prop]_{type}$, $[Prop]_{type}$, $[Prop]_{type})$	$([surB]_{term}$, $[big\_nat\_recr]_{term}$ $[big\_nat\_recl]_{term}$ $[EL]_{term})$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g2	$([move :]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([eq\_refl]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g3	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([subr\_eq0]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g4	$([move/]_{tac}, [->]_{tac}, 0, 0)$	2	$([Prop]_{type}, 0, 0, 0)$	$([eqP]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	1
g5	$([rewrite]_{tac}, 0, 0, 0)$	1	$([Prop]_{type}, 0, 0, 0)$	$([sub0r]_{term}, 0, 0, 0)$	$([=]_{term}, [+]_{term}, [-]_{term})$	0

PROOF STRATEGY 4.9.
Apply case on n.

(1) Prove the base case (a simple task).
(2) Prove the case $0 < n$:
   (a) expand the summation,
   (b) cancel the terms pairwise,
   (c) the terms remaining after the cancellation are the first and the last one.

Using the method presented in [Komendantskaya et al. 2013], if ML4PG was invoked during the proof of Lemma 4.7, it would suggest the lemmas from Example 4.2 and Lemma 4.8. However, 5 irrelevant lemmas about summations would also be suggested (irrelevant in the sense that they do not follow Proof Strategy 4.9). The cluster containing just the two desired lemmas could be obtained after increasing the *granularity* value [Komendantskaya et al. 2013] — a statistical ML4PG parameter that can be adjusted by the user to obtain more precise clusters. The new version of ML4PG suggests four proof fragments, all following Strategy 4.9 without needing to adjust granularity.

The new method brings two improvements: (1) the number of suggestions is increased, but, at the same time, (2) the clusters are more accurate. The proof-patch method considers fragments of proofs that are deep in the proof and were not considered before; therefore, it can find lemmas (more precisely patches of lemmas) that were not included previously in the clusters. In our case study, ML4PG suggests two additional interesting proof fragments. The first one is an intermediate patch of the proof of Lemma 4.8; then, two patches are suggested from this lemma: the proof-patch of the inner sum, and the proof-patch of the outer sum (both of them following Proof Strategy 4.9). The following lemma is also suggested.

LEMMA 4.10. *Let M be a nilpotent matrix, then there exists a matrix N such that $N \times (1 - M) = 1$.*

At first sight, the proof of this lemma is an unlikely candidate to follow Proof Strategy 4.9, since the statement of the lemma does not involve summations. However, inspecting its proof, we can see that it uses $\sum_{i=0}^{n-1} M^i$ as witness for N and then follows Proof Strategy 4.9. In this case, ML4PG suggests the patch from the last five proof-steps that correspond to the application of Proof Strategy 4.9.

The new numbering of features produces more accurate clusters removing the irrelevant lemmas. In particular, using the default settings, ML4PG only suggests the lemma from

Example 4.2, the two patches from Lemma 4.8 and the last patch from Lemma 4.10. If the granularity is increased, the last patch from Lemma 4.10 is the only suggestion — note that this is the closest lemma to Lemma 4.7.

5. EVALUATION

In the previous sections, we have shown how ML4PG has been designed to provide *interesting* and *non-trivial* hints on user's demand, and to be flexible enough to do so at any stage of the proof, and relative to any chosen proof library. However, it is difficult to measure how ML4PG improves the interactive proof building experience, since the usability of a hint varies from user to user. In this section, we present a quantitative method to evaluate how useful ML4PG can be during the proof development.

Machine-learning techniques have been previously used for the automatically generation of proofs in ITPs, see [Duncan 2002; Dixon and Fleuriot 2003; Gransden et al. 2014]. ML4PG was not initially designed with this aim; but we can use the information obtained from clustering to automatically generate proof attempts for a given theorem.

PROOF EXPLORATION METHOD 5.1. *Given the statement of a theorem T and a set of lemmas $L = \{L_i\}_i$, we can use ML4PG to find a proof for T as follows:*

(1) *Using term-clustering and $T \cup L$ as dataset, obtain the cluster C that contains the theorem T (i.e. $C = T \cup \{L_j\}_j$ where $\{L_j\}_j$ is the set of lemmas similar to T).*
(2) *Obtain the sequence of tactics $\{T_1^j, \ldots, T_{n_i}^j\}_j$ used to prove each lemma L_j in C.*
(3) *For each j, try to prove T using $T_1^j, \ldots, T_{n_j}^j$.*
(4) *If no sequence of tactics prove T, then for each tactic use ML4PG to infer the argument for each tactic T_k^j:*
   — *If the argument of T_k^j is an internal hypothesis from the context of a proof, try all the internal hypothesis from the context of the current proof.*
   — *If the argument of T_k^j is an external lemma E, use term-clustering and L as dataset to compute the lemmas in the same cluster as E and try all those lemmas.*
   *** *This can be naturally extended to tactics with several arguments, just trying all the possible combinations.*

Example 5.2. Let T be the lemma `maxnACA` that states the inner commutativity of the maximum of two natural numbers in the SSReflect library:

```
Lemma maxnACA : interchange maxn maxn.
```

and L be the `ssrnat` library of SSReflect. Using Proof Exploration Method 5.1, we can construct a proof of `maxnACA` as follows.

(1) T belongs to the cluster containing the lemmas {`addnACA`, `minnACA`, `mulnACA`} — these 3 lemmas state the inner commutativity of addition, multiplication and the minimum of two naturals respectively.
(2) For this example, we will only consider the proof of the lemma `addnACA` that is proven using the sequence of tactics `by move=> m n p q; rewrite -!addnA (addnCA n).`
(3) The proof of `maxnACA` fails using the sequence of tactics from `addnACA`.
(4) The proof of `addnACA` uses the lemmas `addnA` and `addnCA`. If we cluster these lemmas with the rest of the lemmas of the ssrnat library we find the cluster {`minnA`, `mulnA`, `maxnA`} for `addnA`, and the cluster {`minnAC`, `mulnAC`, `maxnAC`} for `addnAC`.
(5) Using the lemmas `maxnA` and `maxnAC`, we construct the sequence of tactics `by move=> m n p q; rewrite -!maxnA (maxnCA n).` that proves the lemma `maxnACA`.

Using 5 Coq theories of varied sizes, we perform an empirical evaluation of our proof exploration method. Our test data consists of the Basic infrastructure of SSReflect li-

Table V. *Percentage of automatically re-proven theorems.*

Library	Language	Granularity 1	Granularity 3	Granularity 5	Theorems
SSReflect library	SSReflect	36%	35%	28%	1389
JVM	SSReflect	56%	58%	65%	49
Persistent Homology	SSReflect	0%	10%	12%	306
Paths (HoTT)	Coq	92%	91%	94%	80
Nash Equilibrium	Coq	40%	37%	36%	145

brary [Gonthier and Mahboubi 2010], the formalisation about Java-like bytecode presented in [Heras and Komendantskaya 2014b], the formalisation of persistent homology [Heras et al. 2013a], the paths library of the HoTT development [Univalent Foundations Program 2013], and two formalisations of Nash Equilibrium [Vestergaard 2006; Roux 2009]. Using Proof exploration method 5.1, we try to prove every lemma of these libraries with a fully-honest approach (following the terminology from [Kaliszyk and Urban 2014b]): clustering is only performed against the lemmas that have previously proven in the given library. In addition, we study the impact of changing the granularity value.

The results of our experiments can be seen in Table V. The success rate of the proof exploration method depends on how similar are the proofs of theorems in a given library. This explains the high success rate in the `Paths` library, where most of the lemmas are proven using almost the same sequence of tactics, and the low rate in the Persistent Homology library, where just a few lemmas are similarly proven. The granularity value does not have a big impact in the performance of our experiments, and almost the same amount of lemmas are proven with the different values. In some cases, like in the Nash equilibrium library, a small granularity value generates bigger clusters that increase the exploration space allowing to prove more lemmas. However, reducing the granularity value can also have a negative impact; for instance, in the JVM library the number of clusters is reduced and this means a reduction of the proven theorems.

The evaluation method presented in this section shows an estimation of how ML4PG can help during the proof development. This evaluation is a bit artificial since it heavily relies on having a well-developed background theory. For instance, in Example 5.2, the proof exploration method is able to automatically generate a proof for `maxnACA` since the lemmas `maxnA` and `maxnAC` were already in the library, but this is not a common scenario when the user is creating his library. However, ML4PG can be useful for the user even if those two lemmas are not available: ML4PG can suggest that `maxnACA` is similar to `addnACA`, and from the proof of the latter theorem, the user can infer that he needs to state the lemmas `maxnA` and `maxnAC` that were missing in the library. The automatic generation of lemmas based on clustering information was studied for the ACL2 system in [Heras et al. 2013b] — the extrapolation of the technique presented in [Heras et al. 2013b] to Coq is not trivial since ACL2 works with first-order logic and Coq works with higher-order logic.

6. VISUALISATION TOOLS IN ML4PG

In the previous sections, we have seen how ML4PG can be used to find families of proofs following a common pattern, and use those families to automatically prove theorems. However, when the proof-generation method fails, the output provided by ML4PG algorithms is just a set of similar patches with no hints of why these proof-patches are deemed similar. In this section, we present our approach to facilitate the understanding of proof patterns using different visualisation tools that are summarised in Figure 2.

An automaton-like representation of the proof-flow. The first problem that we address is the discovery of the key features that were taken into account during the cluster formation. This is a well-known problem in machine-learning known as *feature selection* [Bishop 2006]. From a given set of features, feature selection algorithms generate different subsets of features and create a ranking of feature subsets. ML4PG uses the *correlation-based feature*

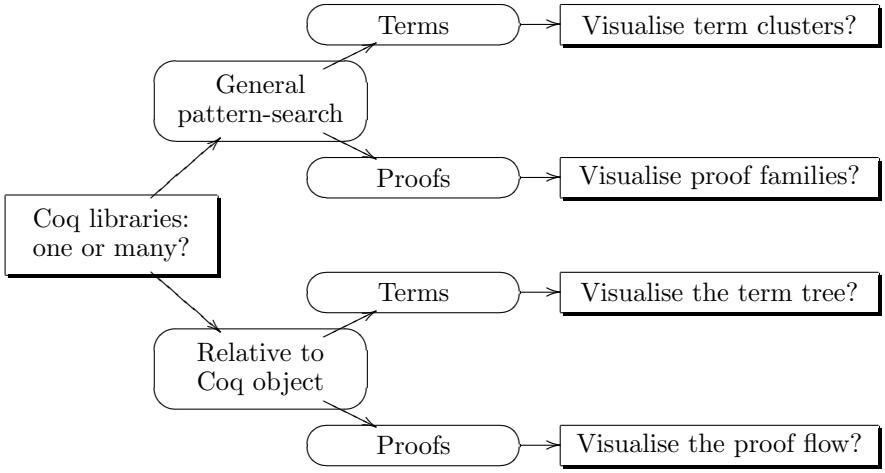

Fig. 2. Visualisation techniques included in ML4PG.

subset selection algorithm implemented in Weka [Hall et al. 2009] — the machine-learning toolbox employed by ML4PG to extract the relevant features.

Example 6.1. In the case study presented at the end of Section 4, the relevant features were:

– The tactics applied in the first and second steps of the proof-patches (`case` and `rewrite` respectively).

– The type of the argument of the tactic applied in the first step of the proof-patches (`nat`).

– The second top symbol of the goal in the second step of the proof-patches (the $\sum$ symbol).

– The first and second auxiliary lemmas applied in the the third step of the proof-patches. The first auxiliary lemma (`sumrB`) is used to expand the summations and is common to all the proofs. There are two different "second" auxiliary lemmas that occur in the proofs of Lemma 4.8 and 4.10 (`big_nat_recr` and `big_nat_recl` — both extract elements of a summation); however, the recurrent feature extraction process have assigned similar values to them.

ML4PG uses this information and produces an automaton-shape representation for discovered proof-patterns and the correlated features that determined the patterns. In our example, the associated "automaton" is depicted in Figure 3.

Generally, given a cluster of proof patches $\mathcal{C}$, we have an automaton A with 5 consecutive states. The ith state of A is labelled with the list of ith goals in the proof-patches contained in $\mathcal{C}$. The transitions between the ith and $i+1$th states are given by the ith tactic of each proof-patch of $\mathcal{C}$. If two or more tactics belong to the same group (see Table III), they are merged in a unique transition; otherwise, the tactics will be shown as different transitions. In addition, each state is annotated with features whose correlation determined the cluster.

A tree representation for terms. As we have previously explained, ML4PG can cluster terms using a recurrent clustering process (see Section 3). When using this tool, the output generated by ML4PG is a list of names of similar terms, and the user should inspect the libraries to compare the different terms. In order to simplify this process, we have created a visualisation tool that generates the ML4PG term-tree (cf. Definition 2.2) given the name associated with a term.

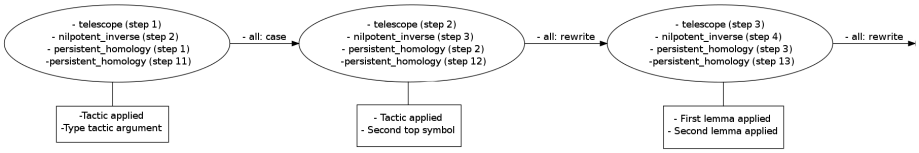

Fig. 3. *Fragment of the automaton corresponding to the cluster of four lemma fragments described in the case study of Section 4* (the whole automaton can be seen in Appendix C). It shows correspondence between certain proof steps of lemmas `telescope` (for the lemma of Example 4.2), `nilpotent_inverse` (for Lemma 4.10), and `persistent_homology` (for Lemma 4.8). Square boxes denote feature correlation where it exists. To be compact, we hide full lemma statements, tactic and auxiliary lemma names, symbols, etc., but they can be shown by ML4PG.

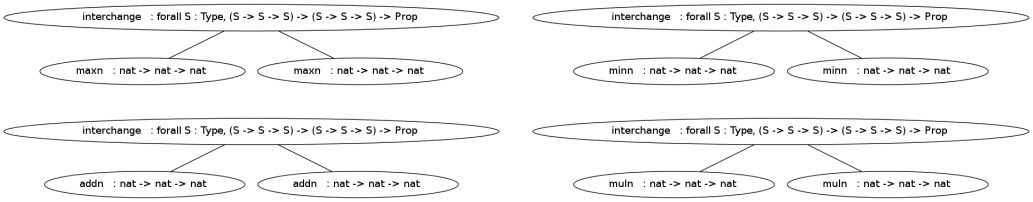

Fig. 4. Term trees associated with `maxnACA` (top left), `minnACA` (top right), `addnACA` (bottom left), `mulnACA` (bottom right).

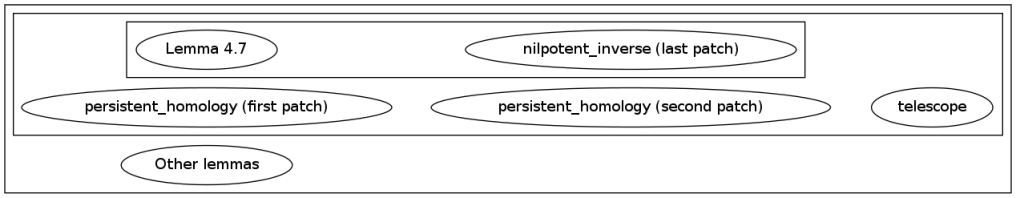

Fig. 5. Cluster visualisation for the case study included at the end of Section 4 regarding the families of similar lemmas to Lemma 4.7: `telescope` (for the lemma of Example 4.2), `nilpotent_inverse` (for Lemma 4.10), and `persistent_homology` (for Lemma 4.8). The outer, middle and inner square contain, respectively, the proof-patches obtained with granularities 1, 3 and 5. It shows correspondence between certain proof steps of lemmas

Example 6.2. In Example 5.2, we have shown how the lemma `maxnACA` is automatically proven thanks to its similarity with lemmas `addnACA`, `minnACA`, `mulnACA`. From Figure 4, it is trivial why these 4 lemmas are similar.

Cluster visualisation. Finally, we have also improved the visualisation of clusters. Initially, ML4PG's output is a list of similar lemmas (or in general terms), and the user can adjust the accuracy of those families varying the granularity value (see the case studied included at the end of Section 4). However, this means that he has to run the clustering process several times, and remember the families that were generated in each iteration. These problems have been tackled thanks to a new visualisation tool that captures the relations between the families of similar proofs/terms obtained using different granularity values, see Figure 5.

7. RELATED WORK

In this section, we compare ML4PG with tools available in Coq and other ITPs, an overview of this comparison is provided in Table VI.

Statistical Proof-Premise Selection in ITPs. Proof automation in ITPs has been enhanced with the connection of this kind of systems with Automated Theorem Provers

Table VI. *Comparison of ML4PG with premise-selection methods, Coq's search mechanisms and dependency graphs.*

	Premise Selection	Search	Dependency graphs	Model Inference	ML4PG
Method	Statistical Proof-Premise Selection	Search	Parsing	Model Inference	Statistical Pattern-recognition
Goal-oriented?	Yes	Yes	Not necessarily	No	Not necessarily
Deterministic?	No	Yes	Yes	No	No
Visual representation	No	No	Yes	Yes	Yes
Covers proofs or terms?	Both	Terms/types	Both	Proofs	Both
Takes into consideration dependencies?	Yes	No	Yes	Yes	Yes
Finds structural similarities beyond concrete syntax?	No	No	No	No	Yes
Finds structurally similar patterns in proofs?	No	No	No	No	Yes
Good for proof automation?	Depend on background theory	No	No	Depend on background theory	Depend on background theory

(ATPs). The workflow of tools like Sledgehammer [Paulson and Blanchette 2010] or HolyHammer [Kaliszyk and Urban 2014a] consists of the following steps: (1) translation of the statement of the theorem (from the ITP format) to a first order format suitable for ATPs; (2) selection of the lemmas (or premises) that could be relevant to prove a theorem; (3) invocation of several ATPs to prove the result; and (4) if an ATP is successful in the proof, reconstruction of the proof in the ITP from the output generated by the ATP.

An important issue in the above procedure is the premise selection mechanism, especially when dealing with big libraries, since proofs of some results can be infeasible for the ATPs if they receive too many premises. Statistical machine-learning methods have been successfully used to tackle this problem in [Urban et al. 2008; Kühlwein et al. 2013; Kaliszyk and Urban 2014a]. In particular, a classifier is constructed for every lemma of the library. Given two lemmas A and B, if B was used in the proof of A, a feature vector $|A|$ is extracted from the statement of A, and is sent as a positive example to the classifier $< B >$ constructed for B; otherwise, $|A|$ is used as a negative example to train $< B >$. Note that, $|A|$ captures statistics of A's syntactic form relative to every symbol in the library; and the resulting feature vector is a sparse (including up to 10^6 features). After such training, the classifier $< B >$ can predict whether a new lemma C requires the lemma B in its proof, by testing $< B >$ with the input vector $|C|$. On the basis of such predictions for all lemmas in the library, this tool constructs a hierarchy of the premises that are most likely to be used in the proof of C.

There are several differences between the statistical premise-selection tools and ML4PG:

— *Aim:* premise selection tools help the prover, trying to increase the number of goals automatically proved by ATPs, and ML4PG assists the user, providing suggestions as proof hints.

— *Features:* premise selection tools extract features from first-order formulas translated from the original syntax of the theorem prover; on the contrary, ML4PG directly works with Coq's higher-order formulas and proofs.
— *Dependencies:* premise selection tools capture the dependencies of lemmas used in the proof of theorems; ML4PG not only captures these dependencies but also information about, for instance, tactics involved in a proof.
— *Machine-learning methods:* Premise selection tools use supervised learning methods, and ML4PG uses unsupervised techniques.
— *Proof Automation:* the success of tools like Sledgehammer depends on several aspects: good methods for premise selection, availability of a useful background theory, and success of ATPs and the proof reconstruction tool. In the case of ML4PG, the availability of a background theory is also a key aspect for success proof automation, but it also depends on the existence of theorems that required similar proofs. Hence, both tools are only as good as the background theory that was previously developed.

ML4PG vs Coq's Searching Tools. Coq/SSReflect already provides comprehensive search mechanisms to inspect the corpus of results available in different libraries. There are several search commands in Coq: `Search` , `SearchAbout`, `SearchPattern` and `SearchRewrite` [Coq development team 2013]. In addition, SSReflect implements its own version of the `Search` command [Gonthier and Mahboubi 2010] — SSReflect's `Search` gathers the functionality of the 4 Coq's search commands. The Whelp platform [Asperti et al. 2006] is a web search engine for mathematical knowledge formalised in Coq, which features 3 functionalities: `Match` (similar to Coq's `Search` command), `Hint` (that finds all the theorems which can be applied to derive the current goal) and `Elim` (that retrieves all the eliminators of a given type).

The existing search mechanisms can be used in two different scenarios. If the user knows how to continue the proof, but he does not remember (or know) the concrete name of the desired auxiliary lemma, it suffices to provide a search pattern to Coq searching engines, e.g. using commands of the form "`Search "distr" in bigop`" or "`Search _ (_ * (\big[_/_]_(_ <- _| _)_))`", where `bigop` is a library, `"distr"` is a pattern in a lemma name, and `_ (_ * (\big[_/_]_(_ <- _| _)_))` is a pattern for search.

In the second scenario, the user needs help in the middle of the proof. Searching mechanisms can be also useful in this case; for instance, using the `Search` command the user can search all the lemmas associated with a concrete term of the current goal. The `Hint` mechanism of Whelp can find all the theorems which can be applied to derive the current goal.

The above search mechanisms are goal directed and deterministic. That is, the user searches the chosen libraries for lemmas related to a concrete type, term or pattern. If the patterns defined by the user are present in the given library, then the user is guaranteed to see the relevant lemmas on the screen.

The situation is more complicated if the user does not know the right pattern to search for. Imagine, for example, being in the middle of constructing a proof, and wishing to get some higher-level hint on how to proceed, wishing there was an ITP expert near, who would suggest a further proof strategy based on his previous experience. ML4PG was created to emulate such intelligent help automatically, using statistical machine-learning algorithms to use the information arising from the "previous experience".

In comparison to the more traditional *search engines*, ML4PG is a goal-independent (unsupervised) tool, i.e. the user does not have to know the required pattern in advance. But, as many statistical machine-learning applications, it is also non-deterministic. That is, the tool failing to suggest a proof pattern does not mean there is no "interesting" pattern to be found. Actually, the notion of a proof pattern being "interesting" is left to the user's judgement, as well.

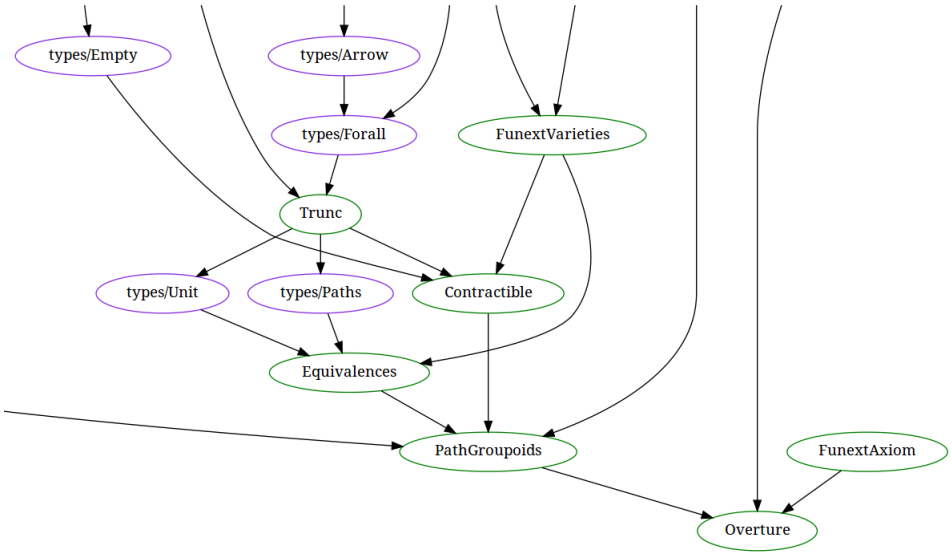

Fig. 6. *Fragment of the dependency graph of the HoTT library.* The complete dependency graph is available at the HoTT wiki[Univalent Foundations Program 2013].

ML4PG vs Coq's Visualisation Tools. Several visualisation tools have been developed to facilitate the understanding of the dependencies that arise in Coq developments. Namely, Coq provides tools to visualise dependencies among files, dependencies between theorems and definitions, proof-trees, and proof-models.

Large Coq developments involve hundreds of files (e.g. 125 files in the proof of the Feit-Thompson theorem); therefore, the computation and visualisation of the dependencies among those files is an important issue to maintain those developments. The Coq distribution provides the `coqdep` tool to compute file dependencies. The output generated by `coqdep` can be fed into the tool developed in [Pottier and Pons 1998] to visualise file dependencies (see Figure 6).

A different set of tools [Pacalet and Bertot 2013; Bertot et al. 2000] has been developed to compute and visualise dependencies between theorems and definitions, see Figure 7. An analysis of proof terms makes it possible to know whether some theorem is used when proving another result or when defining a new concept. Visualising the complete graph of dependence is useful to detect the results that are not used, and to foresee the impact of modifying a theorem or definition.

These dependency graphs are designed to, mainly, work with proofs rather than object definitions and do not show the term structure per se, but only the dependencies between the auxiliary lemmas/constructors used. Moreover, they give a complete information of all the results that were necessary to prove a given lemma, making hard to read them, due to the presence of this complete information. Hence, statistical machine learning may be useful for data-mining the above information in order to discriminate unimportant features of the lemma, and highlight those that are important. ML4PG essentially does this. ML4PG's recurrent clustering captures term and lemma dependencies recurrently, via the feature extraction which itself involves recurrent clustering of all previously defined terms.

ML4PG's proof clustering in particular is very close to dependency graphs for lemmas, in that both are capturing mutual or inductive dependencies of all lemmas needed to prove the given statement. As a result, ML4PG can be seen as a post-processing tool for dependency graphs. Given a big graph as shown in Figure 7, what is the right way to discriminate its unimportant features? How to decide which of them are important or not? One of the ways

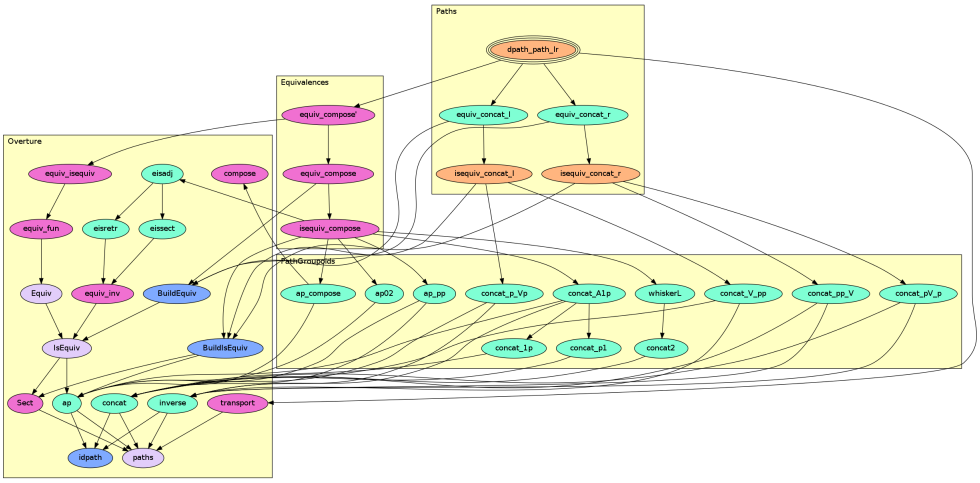

Fig. 7. *Dependency graph for the theorem* `dpath_path_lr` *included in the* `Paths` *library of HoTT.*

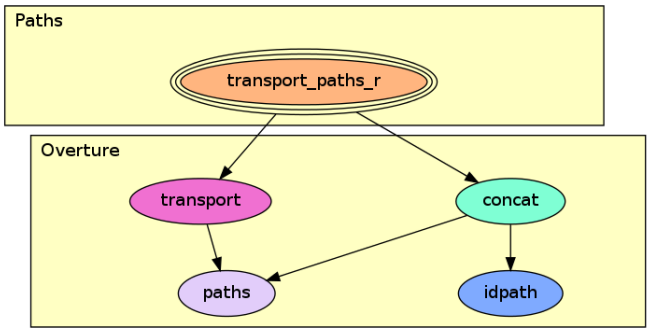

Fig. 8. *Dependency graph for the definition of* `transport_paths_r` *included in the* `Paths` *library of HoTT.*

to do this, is to determine that relatively to other lemmas in the library. This is achieved by applying clustering in ML4PG. Taking all of the above features into account, ML4PG can associate various object definitions and lemma statements and lemma proofs.

The tools presented in [Pacalet and Bertot 2013; Bertot et al. 2000] can also be used to visualise dependency graphs for terms, see Figure 8. ML4PGs term-structure clustering goes beyond these dependency graphs for terms, and carefully captures the tree term shape, structure, and dependency on other term- and type- structures present in the library (see Figure 4). ML4PG's approach makes sure that dependency of auxiliary terms/types on other terms/types defined in Coq libraries is taken into account recurrently.

Coq also supports the visualisation of the proof-tree during an interactive proof development using the Prooftree tool [Tews 2011]. This tool was aimed to visualise the different subgoals that arise during the development of a proof, see Figure 9. ML4PG automaton representation of proofs (cf. Figure 3) not only shows the applied tactics, but also the generated-subgoals in each step, and compare the proof flow of the current proof with the steps followed in other theorems.

Finally, the tool presented in [Gransden et al. 2014] can infer models from Coq developments. Those models are represented as automata that are latter used to generate proof attempts. Since the models were designed with the aim of automatically generate proofs,

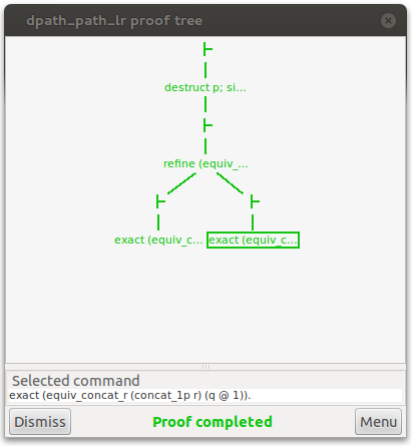

Fig. 9. *Proof tree for the theorem* `dpath_path_lr` *included in the* `Paths` *library of HoTT.*

they are not human-readable. ML4PG's graphical representation of automata is simpler than the work presented in [Gransden et al. 2014].

8. CONCLUSIONS

We have presented several enhancements of the ML4PG system. *Term clustering* adds a new functionality to ML4PG: the user can receive suggestions about families of similar definitions, types and lemma shapes (in fact, any Coq terms). The *proof-patch method* is employed to analyse the properties of the patches that constitute a proof. The whole syntax of Coq libraries is now subject to *recurrent clustering*, which increases the number and accuracy of families of similar proofs suggested by ML4PG. In addition, this information is taken into account in an automatic proof-generation method. Finally, the different visualisation tools implemented in ML4PG facilitates the interpretation of clusters of similar proof-patches and terms.

Further improvements in *accuracy* (e.g. including proof contexts into the analysis) and *conceptualisation* for clustered terms and proofs are planned. The families of similar proofs and terms can be the basis to apply symbolic techniques to, for instance, infer models from proof traces [Gransden et al. 2014], or generate auxiliary results using mutation of lemmas [Heras et al. 2013b]. We expect that the incorporation of these techniques help in the goal pursued by ML4PG: make the proof development easier.

REFERENCES

A. Asperti and others. 2006. A Content Based Mathematical Search Engine: Whelp. In *TYPES'04 (LNCS)*, Vol. 3839. 17–32. DOI:http://dx.doi.org/10.1007/11617990_2

Y. Bertot, O. Pons, and L. Pottier. 2000. *Dependency Graphs for Interactive Theorem Provers*. Technical Report 4052.

C. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

Coq development team. 2013. *The Coq Proof Assistant Reference Manual, version 8.4pl3*. Technical Report.

T. Coquand and G. Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2/3 (1988), 95–120. DOI:http://dx.doi.org/10.1016/0890-5401(88)90005-3

T. Coquand and C. Paulin-Mohring. 1990. Inductively defined types. In *Colog'88 (LNCS)*, Vol. 417. 50–66.

L. Dixon and J. D. Fleuriot. 2003. IsaPlanner: A Prototype Proof Planner in Isabelle. In *19th International Conference on Automated Deduction (CADE'2003) (LNCS)*, Vol. 2741. 279–283. DOI:http://dx.doi.org/10.1007/978-3-540-45085-6_22

H. Duncan. 2002. *The use of Data-Mining for the Automatic Formation of Tactics*. Ph.D. Dissertation. University of Edinburgh.

F. Garillot and others. 2009. Packaging mathematical structures. In *TPHOLs'09 (LNCS)*, Vol. 5674.

G. Gonthier and others. 2013. A Machine-Checked Proof of the Odd Order Theorem. In *ITP'13 (LNCS)*, Vol. 7998. 163–179.

G. Gonthier and A. Mahboubi. 2010. An introduction to small scale reflection. *Journal of Formalized Reasoning* 3, 2 (2010), 95–152. DOI:http://dx.doi.org/10.6092/issn.1972-5787/1979

T. Gransden, N. Walkinshaw, and R. Raman. 2014. Mining State-Based Models from Proof Corpora. In *Conferences on Intelligent Computer Mathematics (CICM'14) (LNCS)*, Vol. 8543. 282–297.

M. Hall and others. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explorations* 11, 1 (2009), 10–18. DOI:http://dx.doi.org/10.1145/1656274.1656278

J. Heras and others. 2013a. Computing Persistent Homology within Coq/SSReflect. *ACM Transactions on Computational Logic* 14, 4 (2013).

J. Heras and others. 2013b. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *LPAR-19 (LNCS)*, Vol. 8312. 389–406.

J. Heras and E. Komendantskaya. 2012–2014a. ML4PG: downloadable programs, manual, examples. (2012–2014). www.computing.dundee.ac.uk/staff/katya/ML4PG/.

J. Heras and E. Komendantskaya. 2013. ML4PG in Computer Algebra Verification. In *CICM'13 (LNCS)*, Vol. 7961. 354–358.

J. Heras and E. Komendantskaya. 2014b. Recycling Proof Patterns in Coq: Case Studies. *Journal Mathematics in Computer Science, accepted* (2014).

C. Kaliszyk and J. Urban. 2013. Lemma Mining over HOL Light. In *LPAR-19 (LNCS)*, Vol. 8312. 503–517.

C. Kaliszyk and J. Urban. 2014a. Learning-Assisted Automated Reasoning with Flyspeck. *Journal of Automated Reasoning* 53, 2 (2014), 173–213.

C. Kaliszyk and J. Urban. 2014b. Learning-assisted Theorem Proving with Millions of Lemmas. *CoRR* abs/1402.3578 (2014).

E. Komendantskaya and others. 2013. Machine Learning for Proof General: interfacing interfaces. *Electronic Proceedings in Theoretical Computer Science* 118 (2013), 15–41.

D. Kühlwein and others. 2013. MaSh: Machine Learning for Sledgehammer. In *ITP'13 (LNCS)*, Vol. 7998. 35–50.

A. Pacalet and Y. Bertot. 2009–2013. The DpdGraph tool. (2009–2013). https://anne.pacalet.fr/dev/dpdgraph/.

L. C. Paulson and J. C. Blanchette. 2010. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In *8th International Workshop on the Implementation of Logics (IWIL'10)*. DOI:http://dx.doi.org/10.1.1.186.3278

L. Pottier and O. Pons. 1998. *dependtohtml: Creating Hypertext graphical representation of directed graphs*. Technical Report.

S. Le Roux. 2009. Acyclic Preferences and Existence of Sequential Nash Equilibria: A Formal and Constructive Equivalence. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'07) (LNCS)*, Vol. 5674. 293–309.

H. Tews. 2011. Automatic Library Compilation and Proof-Tree Visualisation for Coq Proof General. In *3rd Coq Workshop*. http://askra.de/software/prooftree/

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study. https://github.com/HoTT/HoTT/wiki.

J. Urban and others. 2008. MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance. In *IJCAR'08 (LNCS)*, Vol. 5195. 441–456. DOI:http://dx.doi.org/10.1007/978-3-540-71070-7_37

R. Vestergaard. 2006. A constructive approach to sequential Nash equilibria. *Information Processing Letter* 97 (2006), 46–51.

A. FORMULA FOR GALLINA TOKENS

We split Gallina tokens into the following groups.

— Group 1: `forall`, `->`.
— Group 2: `fun`,
— Group 3: `let`, `let fix`, `let cofix`.
— Group 4: `fix`, `cofix`.
— Group 5: `@`,
— Group 6: `match`, `if`.
— Group 7: `:=`, `=>`, `is`.
— Group 8: `Inductive`, `CoInductive`.
— Group 9: `exists`, `exists2`.
— Group 10: `:`, `:>`, `<:`, `%`.

The formula for the jth Gallina token of the nth group is given by the formula

$$-(n + \sum_{i=0}^{j} \frac{1}{10 \times 2^{i-1}})$$

B. GROUPS OF COQ TACTICS AND NUMBER ASSIGNMENT

Table VII splits the Coq tactics into different groups. The formula used in the function $[.]_{tactic}$ to compute the value of a Coq tactic is given by $i + \sum_{j=0}^{k} \frac{1}{10 \times 2^{j-1}}$ where i is the group of the tactic and k is the position of the tactic in that group (cf. right side of Table VII).

Table VII. *Groups of Coq tactics*

Group	Tactics of the group
Group 1: Applying theorems	exact, eexact, assumption, eassumption, refine, apply, eapply, simple apply, lapply,
Group 2: Managing inductive constructors	constructor, split, exists, left, right, econstructor, esplit, eexists, eleft, eright
Group 3: Managing local context	intro, intros, clear, rever, move, rename, set, remember, pose, decompose
Group 4: Controlling proof flow	assert, cut, pose, specialize, generalize, evar, instantiate, admit, absurd, contradition, contradict, exfalso
Group 5: Case analysis and induction	destruct, case, ecase, simple destruct, induction, einduction, elim, eelim, simple induction, double induction, dependent induction, functional induction, discriminate, injection, fix, cofix, case_eq, elimtype
Group 6: Rewriting expressions	rewrite, erewrite, cutrewrite, replace, reflexivity, symmetry, transitivity, subst, stepl, change
Group 7: Performing computations	cbv, compute, vm_compute, red, hnf, simpl, unfold, fold, pattern, conv_tactic
Group 8: Automation	auto, trivial, eauto, autounfold, autorewrite
Group 9: Decision procedures	tauto, intuition, rtauto, firstorder, congruence
Group 10 Equality	decide equality, compare, simplify_eq, esimplify_eq
Group 11 Inversion	inversion, dependent inversion, functional inversion, quote
Group 12 Classical tactics	classical_left, classical_right
Group 13 Automatizing	omega, ring, field, fourier
Group 14	Rest of Coq tactics

C. AUTOMATON

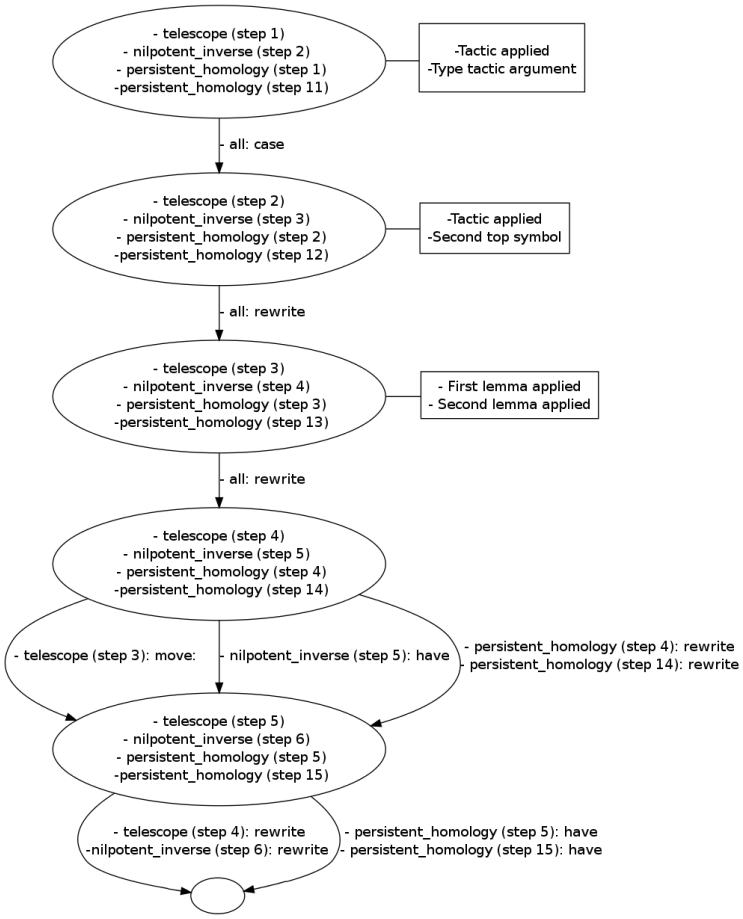

Fig. 10. *Automaton corresponding to the proof cluster of four lemma fragments described in the case study of Section 4.* The automaton shows the five proof steps, of which the first three are shown to influence the cluster formation. It uses Lemma names: `telescope` for the lemma of Example 4.2, `nilpotent_inverse` for Lemma 4.10, and `persistent_homology` for Lemma 4.8. In addition to Lemma names, ML4PG can show lemma statements, and it can provide details of the "Tactic applied", "Type tactic argument", "Second top symbol", "First/Second lemma applied" fields.