Recycling Proof Patterns in Coq: Case Studies

Jónathan Heras and Ekaterina Komendantskaya

Abstract. Development of Interactive Theorem Provers has led to the creation of big libraries and varied infrastructures for formal proofs. However, despite (or perhaps due to) their sophistication, the re-use of libraries by non-experts or across domains is a challenge. In this paper, we provide detailed case studies and evaluate the machine-learning tool ML4PG built to interactively data-mine the electronic libraries of proofs, and to provide user guidance on the basis of proof patterns found in the existing libraries.

Keywords. Interactive Theorem Proving, Coq, SSReflect, Machine Learning, Clustering.

1. Introduction

Interactive theorem provers (ITPs) (e.g. Agda [10], Coq [12], Isabelle/HOL [30], Matita [3] to name a few) are a family of higher-order languages allowing the formalisation of a wide variety of domains, ranging from mathematical theories to software verification. The most recent achievements concerned formalisation and computer verification of results coming from Group Theory [15], Topology [19], Real Numbers [26], Discrete Mathematics [17] and Security [1]. The successful and efficient ITP programming often requires a combination of mathematical and programming intuition; see e.g. [6]. The use of a rich higher-order language implies that there can be a rich variety of approaches to the formalisation and proof development for a given task. Thus, a programmer relies on the previous experience and ability to *"creatively"* adapt already used proof techniques and *patterns* in newly constructed proofs. This explains why a "steep learning curve" is often mentioned as one of the big obstacles to wider adoption of ITPs. In this paper, we are probing the abilities of our recent machine-learning tool ML4PG [20, 25] to find interesting proof patterns automatically, and thus enable a more efficient use of ITPs by specialists coming from a wider range of domains.

The work was supported by EPSRC grant EP/J014222/1 and EP/K031864/1.

Development of 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 [15]). 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 reusable concepts and proof ideas.

A different, but related, challenge is faced during the creation of these libraries. These frameworks are developed by teams (e.g. 15 people were involved in the Feit-Thompson theorem project), and the situation is similar in industry where teams use ITPs to verify the correctness of hardware and software systems. In those teams, each user has his own definitions, notation and proof-style, which makes the collaborative proof development difficult. In both scenarios, it would be extremely helpful to use a tool that could detect patterns across different users, notation and libraries.

To address these challenges, we propose ML4PG – a machine-learning extension to the Proof General [4] interface for Coq [7] and its SSReflect dialect [16]. Our main goal is to prove the concept: *it is possible to embed a lightweight statistical machine-learning tool into an ITP proof interface, and use it interactively to find non-trivial patterns in existing proofs and aid new proof developments.*

The ML4PG package for Proof General features the following main functions:

- The user works within the interactive environment of Coq/SSReflect, and has an option to call ML4PG from the Proof General interface whenever he needs to find some proof patterns.
- Based on the user's choice, ML4PG compiles the chosen libraries, and extracts significant proof features from the existing lemmas and proofs;
- ML4PG connects to machine-learning tools, and runs a number of experiments on clustering the data for each user query. Based on the results, it chooses the most reliable patterns; thus relieving the Coq programmer of the laborious step of post-processing the statistical results.
- If the user chooses to see only patterns related to his current proof goal, ML4PG would further filter the results and show the families of related proofs to the user.

Section 2 gives an overview of ML4PG features, details of its implementation are given in [25]. ML4PG's features have been substantially extended since [25], we will briefly survey the changes in the Conclusions and Future Work section. In this paper, we do not focus on the ML4PG implementation *per se*, although we use it for all the examples and experiments shown in this paper. Our main goal here is to show how useful the automated proof pattern detection can be in different domains.

To illustrate this, we devise three experiments ("user scenarios") to test ML4PG. Each example is designed to demonstrate a different aspect of proof-pattern recognition. To demonstrate ML4PG's ability to adapt to different domains, we deliberately illustrate each user scenario by using libraries coming from different subject areas, ranging from basic mathematical infrastructures to software verification.

User Scenario 1 illustrates how to use ML4PG for detecting proof patterns prior to the start of a new proof development. To achieve this, Section 3 analyses fundamental libraries that are common in most developments using the SSReflect library [16]. The SSReflect library was developed as the infrastructure for the formalisation of the Four Colour Theorem [14] and has played a key role in the formal proof of the Feit-Thompson theorem [15]. Up to version 1.4, the SSReflect library was distributed together with the MathComp library (that contains the theories about the development of the proof of the Feit-Thompson theorem); from version 1.5, the SSReflect library can be downloaded independently from the MathComp library.

In this first scenario, we use pattern recognition with the aim of spotting common proof patterns across fundamental libraries (1404 theorems). The benefits of using ML4PG in this context is that it can be used to speed up the beginning of a proof development, making it easier to recycle patterns already available in the libraries.

User Scenario 2 considers the problem of proof-pattern discovery in a different light. In User Scenario 1, there was always an interesting underlying proof pattern hidden in the big proof libraries, "waiting" to be discovered. What if, despite the user's hope that one library may contain similar proof strategies to another, the actual proofs are in fact too different to be recycled? Section 4 studies the results that ML4PG obtains working with two different Coq libraries formalising results from game theory [32, 33]. One might hope that they contain similar proof patterns, since they formalise the same subject domain; but in fact, ML4PG shows that the actual proof strategies used in [33] and [32] are completely different. This "negative" output given by ML4PG may in reality save the user's time inspecting these libraries manually.

User Scenario 3 considers the situation when a team of several people develops a set of different modules within one bigger (industrial-scale) verification effort, see Section 5. For this purpose, we translate the proofs of correctness of the Java Virtual Machine (JVM) given in [23] into Coq. Industrial scenario of interactive theorem proving may differ significantly from the academic scenarios above. Namely, industrial verification tasks often feature a bigger number of routine cases and similar lemmas; and also such tasks are distributed across a team of developers. Here, the inefficiency of automated proving often arises when programmers use different notation to accomplish very similar tasks, and thus a lot of work gets duplicated, see also [11]. We tested ML4PG in exactly such scenario: we assumed that a programming team has collectively developed proofs of *a) soundness of specification, and b) correctness of implementation* of Java byte code for

a dozen of programs computing multiplication, powers, exponentiation, and other functions. We assumed that there is a relative novice in the team, trying to "learn" from the previous team efforts, in order to repeat their proof steps for a new Java function (factorial in our setting). He calls ML4PG, which discovers common patterns among these proofs and relevant lemmas (around 150 training examples in total). The suggested clusters indeed helped to advance the proofs of properties a) and b) for the Java byte code of the factorial function.

This is the first thorough and systematic evaluation of ML4PG, note that [25] focused mainly on the user interface and contained very simple examples. The case studies presented here convince us that when ML4PG statistically discovers *proof clusters*, it does actually find meaningful, non-trivial and interesting patterns in proofs across different libraries, theories and users. This kind of proof analysis can speed up the proof development by suggesting reusable proof strategies. ML4PG works on the background of Proof General, and if called, provides clustering results almost instantly; thus, can be used interactively, as a handy tool on request. Finally, it may be used for educational purposes, as automated proof-pattern recognition may help to smooth the learning curve, see User Scenario 3.

ML4PG and all examples presented in this paper are available in [20].

Related Work. ML4PG's originality is two-fold, as it can be compared to alternative methods of using machine-learning in automated theorem proving, as well as to Coq/SSReflect tools allowing interactive pattern-search.

Related work on using machine-learning in ITPs concerned hints in lemma generation for Isabelle/HOL [24], proof strategy discovery in Isabelle/HOL [5], speed up in proof automation in Mizar [27] and statistical tactic analysis in Isabelle [13]. Comparing to these tools, we use unsupervised, rather than supervised, learning; and we do not use sparse machine-learning methods. (See also [21,25] for a detailed comparison of different machine learning tools applied in various theorem provers.) We do not have a quantitative target when it comes to improving *interactive* proof building experience: neither speed up in automated proof search nor the number of automatically proven theorems are the main criteria of success. Instead, the user experience is the main parameter we target. We generally follow the "qualitative" intuition that ML4PG, being an interactive hint generator, must provide interesting and non-trivial hints on user demands, and should be flexible and fast enough to do so in real time, at any stage of the proof, and relative to any chosen proof library.

Comparing to some of the above approaches, ML4PG does not only analyse the lemma statements, but also involves user tactics and user-defined proof-steps into the statistical proof-pattern recognition process. This feature also makes ML4PG sensitive (or *adaptable*) to proof styles innate to a particular user, research community, or subject area (cf. Sections 3–5). In illustration of this point, User Scenario 3 and Section 5 consider cases when different lemma statements have similar proofs; User Scenario 2 and Section 4 discuss cases when similar lemma statements require a completely different proof strategy.

Comparing to symbolic methods of proof pattern search in Coq, e.g. `Search`, `SearchPattern`, `SearchAbout`, `SearchRewrite` [12, 16] and `Whelp` [2], ML4PG's originality is in introducing statistical pattern-recognition into the rich family of existing searching mechanisms in Coq/SSReflect. Unlike symbolic pattern-search, ML4PG can discover "unexpected" proof-patterns that go beyond the patterns the user would try as a searching template when using symbolic pattern-search facilities. Whereas the existing Coq searching tools try to match the user-provided template with other lemma/theorem *statements*, ML4PG takes into consideration the *proof* statistics in conjunction with the lemma shapes. These two features – pattern-search without a pre-defined template and the attention to the various proof parameters – allow to achieve results often orthogonal to the symbolic pattern search. Section 5 illustrates this new view on proof-pattern search.

2. ML4PG

In this section, we present the main functionality that ML4PG offers to the user. ML4PG works with Coq and its SSReflect dialect, and it does not assume any machine-learning knowledge from the user. The guidance it provides may come in different forms. The user may prefer the statistical hint to be related to the current proof-step (cf. User Scenario 3), or give information about proof-patterns arising in a library irrespective of the current proof-step (cf. User Scenarios 1 and 2). The user may choose to data-mine only the current library, or a number of proof-libraries coming from different domains or different users. Finally, the user may wish to experiment with proof clusters of different sizes or with different machine-learning algorithms, see Table 4. These choices are accommodated within ML4PG, see [25] for a detailed description of the user interface.

ML4PG functionality is achieved in the following way.

F.1. it works on the background of Proof General extracting some low-level features from proofs in Coq/SSReflect.

F.2. it automatically sends the gathered statistics to a chosen machine-learning interface and triggers execution of a clustering algorithm according to the choice of the user;

F.3. it does some post-processing of the results given by the machine-learning tool, and displays families of related proofs to the user.

Stage **F.1** is devoted to collecting statistics from proofs. The discovery of statistically significant features in data is a research area of its own in machine-learning, known as *feature extraction*, see [8]. Statistical machine-learning algorithms classify given examples seen as points in an m-dimensional space, where m is the maximum number of features each example may be characterised by. Irrespective of the particular feature extraction algorithm used, most pattern recognition tools [8] will require that the number of selected features is limited and fixed – the exception to this is a special class of methods called "sparse" methods [9].

ML4PG has its own feature extraction method that collects statistics from the interaction between the user and the prover. The feature extraction is done at the time of the interactive proof construction in the current library or during the Coq compilation for an external library. The feature extraction method captures information from proofs based on the correlation of a few chosen parameters within five proof steps. For each proof step, the parameters are:

1-2 the names and the number of tactics used in one command line,
3 types of the tactic arguments;
4 relation of the tactic arguments to the (inductive) hypotheses or library lemmas,
5-7 three top symbols in the term-tree of the current subgoal, and
8 the number of subgoals each tactic command-line generates.

When the correlation of these few parameters is taken within a few proof-steps, the arising statistics reveal patterns that can tell a lot about the "meta" proof strategy expressed by the tactics and subgoals. The details and discussion of this feature-extraction method can be found in [25], and the new experimental extensions of it are available in [20]. We will not focus on the technical details of the ML4PG feature extraction here, but rather concentrate in the coming sections on proving the point that these simple statistical parameters (40 for one proof patch of five possibly composite proof steps) can indeed capture some essential proof-strategies, interesting and helpful enough from the user's perspective.

Once all features are extracted, ML4PG is ready to communicate with machine learning interfaces (Stage **F.2**). ML4PG is built to be modular – that is, the feature extraction is first completed within the Proof General environment, where the data is gathered in the format of hash tables, and then these tables are converted to the format of the chosen machine-learning tool. In [25], we connected ML4PG to several machine-learning algorithms available in Matlab [29] and Weka [18]; the results that we obtained with both systems were similar and Weka has the advantage of being an open-source software; hence, we use only Weka throughout this paper, but see [25] for a discussion of Matlab facilities.

ML4PG offers a choice of pattern-recognition algorithms. ML4PG is connected only to clustering algorithms [8] – a family of unsupervised learning methods. Unsupervised learning is chosen when no user guidance or class tags are given to the algorithm in advance: in our case, we do not expect the user to "tag" the library proofs in any way. Clustering techniques divide data into n groups of similar objects (called clusters), where the value of n is provided by the user. There are several clustering algorithms available in Weka (K-means, FarthestFirst and Expectation Maximisation, in short E.M.) and the user can select the algorithm using the ML4PG menu included in the Proof General interface. We illustrate the effect of changing clustering algorithms in Table 4.

As will be illustrated in the later sections, various numbers of clusters can be useful: this may depend on the size of the Coq library, and on existing similarities between the proofs. ML4PG has its own algorithm that determines the optimal

number of clusters interactively, and based on the library size. As a result, the user does not provide the value of n directly, but just decides on *granularity* in the ML4PG menu, by selecting a value between 1 and 5, where 1 stands for a low granularity (producing fewer large clusters) and 5 stands for a high granularity (producing many smaller clusters). Given a granularity value g, the number of clusters n is given by the formula

$$n = \lfloor \frac{\text{objects to cluster}}{10 - g} \rfloor.$$

It is worth mentioning that it is the nature of statistical methods to produce results with some probability, and not being able to provide guarantees that a certain cluster will be found for a certain library. However, ML4PG ensures quality of the output in several different ways (Stage **F.3**). First of all, the results are not taken from one random run of a clustering algorithm – instead, ML4PG output shows a digest of clustering results coming from 200 runs of the clustering algorithm. The 200 runs were experimentally found to be optimal for noticing important statistics in ML4PG setting. Only clusters that appear frequently enough are displayed to the user. There is a way to manipulate the frequency threshold within ML4PG. Another measure is a *proximity value* assigned by clustering algorithms to every term in a cluster – the value ranges from 0 to 1, and indicates the certainty of the given example belonging to the cluster. This proximity value is also taken into account by ML4PG before the results are shown. If a lemma is contained in several clusters, proximity and frequency values are used to determine one "most reliable" cluster to display.

We refer to the ML4PG user manual [20] for a more detailed description of how to use the tool.

3. User scenario 1. Detecting patterns in early-stages of the development

Users of ITPs usually start their developments loading some libraries. Those libraries contain definitions, lemmas and theorems that will be used as background theory during the proof process. Some of those libraries are specific for concrete theories, but others are common for almost every development. The common libraries contain strategies and definitions that can be extrapolated to other contexts; however, detecting lemmas that follow a concrete proof-strategy can be a challenge. In this first scenario, we study the patterns that appear in the SSReflect library [16].

The second purpose of this section is to set terminology and the general style of statistical proof-pattern analysis we will use throughout other sections.

The SSReflect library extends the Coq proof language and consists of 7 files containing basic theories about: natural numbers, lists, booleans, functions, finite types, choice types and types with a decidable equality. The library contains a

total of 1404 theorems; therefore, a manual inspection of these theorems to detect patterns is unfeasible. In our first scenario, we test how ML4PG can be used to detect patterns in the SSReflect library.

We analyse clusters that are produced in the SSReflect library using the K-means algorithm and the value 5 as granularity parameter, these options produce the best results in our experiments. ML4PG discovers 280 clusters using those parameters. In the 45% of those clusters (126 clusters), all the lemmas belong to the same library. We call a cluster *homogeneous* if it contains lemmas and theorems from one library, and *heterogeneous* if it contain objects from different libraries.

The mean size of these homogeneous clusters are 4 elements, and the similarities of the lemmas of a cluster can be easily spotted in most of the cases. From the 126 clusters, we can obtain the following classification of clusters.

- 36% of the clusters consists of lemmas about related functions.

  Example 3.1. Examples of this kind of clusters are the ones including lemmas about: `take` and `drop` (`take` takes the first n elements of a list and `drop` removes the first n elements of the list):

  ```
  Lemma map_take s : map (take n0 s) = take n0 (map s).
  Lemma map_drop s : map (drop n0 s) = drop n0 (map s).
  ```

- 20% of clusters contain lemmas that follow the same proof structure and that share some common auxiliary results.

  Example 3.2. Examples of this kind of cluster appears in several libraries, for instance in the `seq` library:

  ```
  Lemma has_map a s : has a (map s) = has (preim f a) s.
  Proof. by elim: s => //= x s ->. Qed.

  Lemma all_map a s : all a (map s) = all (preim f a) s.
  Proof. by elim: s => //= x s ->. Qed.

  Lemma count_map a s : count a (map s) = count (preim f a) s.
  Proof. by elim: s => //= x s ->. Qed.
  ```

- 13% of clusters consists of theorems that are used in the proofs of other theorems of the same cluster.

  Example 3.3. ML4PG discovers that the following two lemmas are in the same cluster:

  ```
  Lemma altP : alt_spec b.
  Lemma boolP : alt_spec b1 b1 b1. Proof. exact: (altP idP). Qed.
  ```

- 11% of clusters are formed by view lemmas, an important kind of lemmas that are used in SSReflect to apply boolean reflection [16].

Example 3.4. ML4PG finds a cluster with the following two view lemmas coming from the `fintype` library:

```
Lemma unit_enumP : Finite.axiom [::tt]. Proof. by case. Qed.
Lemma bool_enumP : Finite.axiom [:: true; false].
Proof. by case. Qed.
```

- 5% of the clusters contain equivalence lemmas that are proven just by simplification.

  Example 3.5. An example of this kind of clusters is given by the cluster that contains the following lemmas:

  ```
  Lemma multE : mult = muln. Proof. by []. Qed.
  Lemma mulnE : muln = muln_rec. Proof. by []. Qed.
  Lemma addnE : addn = addn_rec. Proof. by []. Qed.
  Lemma plusE : plus = addn. Proof. by []. Qed.
  ```

- 4% of the clusters consists of lemmas that are solved using analogous lemmas.

  Example 3.6. An example of clusters that consists of lemmas that are solved using analogous lemmas is the one containing the following two lemmas.

  ```
  Lemma addnAC : right_commutative addn.
  Proof. by move=> m n p; rewrite -!addnA (addnC n). Qed.

  Lemma subnAC : right_commutative subn.
  Proof. by move=> m n p; rewrite -!subnDA addnC. Qed.
  ```

  Namely, lemma `subnDA` (`forall (a b c : nat), a - (b + c)= (a - b) - c`) can be obtained automatically from lemma `addnA` (`forall (a b c : nat), a + (b + c)= a + b + c`) using techniques like lemma analogy [22].

In the case of heterogeneous clusters (clusters that include lemmas from different libraries), ML4PG discovers 154 clusters. In this case, the size of the clusters is bigger than in the case of homogeneous clusters; namely, the mean size is 8 lemmas per cluster. The different clusters can be classified as follows.

- 31% of the clusters contain lemmas that state properties applicable to several operators from different libraries.

  Example 3.7. ML4PG discovers a cluster containing lemmas about the associativity of the addition of natural numbers (`addn` function) and the associativity of the concatenation of lists (`++` operator).

  ```
  Lemma catA s1 s2 s3 : s1 ++ s2 ++ s3 = (s1 ++ s2) ++ s3.
  Proof. by elim: s1 => //= x s1 ->. Qed.

  Lemma addnA : associative addn.
  Proof. by move=> m n p; rewrite (addnC n) addnCA addnC. Qed.
  ```

- 27% of the clusters consists of lemmas related to operations over base case of types.

  Example 3.8. As an example of this kind of clusters, ML4PG discovers that there is a strong correlation among the following four lemmas:

```
Lemma andTb : left_id true andb. Proof. by []. Qed.
Lemma orFb : left_id false orb. Proof. by []. Qed.
Lemma mul0n : left_zero 0 muln. Proof. by []. Qed.
Lemma sub0n : left_zero 0 subn. Proof. by []. Qed.
```

- 12% of the clusters come from lemmas whose proof rely on the fundamental lemmas.

  Example 3.9. ML4PG discovers a cluster with the following two lemmas about `rot` (that rotates a list `l` left `n` times) and the `expn` (exponentiation function).

```
Lemma rot0 s : rot 0 s = s.
Proof. by rewrite /rot drop0 take0 cats0. Qed.

Lemma expn_eq0 m e : (m ^ e == 0) = (m == 0) && (e > 0).
Proof. by rewrite !eqn0Ngt expn_gt0 negb_or -lt0n. Qed.
```

  At first sight, it seems that the only similarity between these two lemmas is that they only use rewriting rules in their proofs, however if we carefully inspect the lemmas that are used for rewriting, we notice that most of them are fundamental lemmas about `nil` (the base constructor for the `list` type) and `0` (the base constructor for the `nat` type).

- 9% of the clusters combine lemmas from the libraries about lists and natural numbers – note that the definition of lists and natural numbers is quite similar, both have one base case and a recursive one, so several lemmas are solved applying induction and using the inductive hypothesis.

  Example 3.10. An example of this kind of clusters is given by the one that consists of the following lemmas:

```
Lemma catrev_catr s t u : catrev s (t ++ u) = catrev s t ++ u.
Proof. by elim: s t => //= x s IHs t; rewrite -IHs. Qed.

Lemma mulnDl : left_distributive muln addn.
Proof.
  by move=> m1 m2 n; elim: m1 => //= m1 IHm;
  rewrite -addnA -IHm.
Qed.
```

```
Lemma mem_cat x s1 s2:
    (x \in s1 ++ s2) = (x \in s1) || (x \in s2).
Proof. by elim: s1 => //= y s1 IHs; rewrite !inE /= -orbA -IHs.
Qed.
```

In all these lemmas, we can see that induction is applied and after the use of some rewriting rules the inductive hypothesis is applied to finish the proof.

The similarity of most clusters (83% of them) can be easily discovered just inspecting the statement of the lemmas and their proofs. However, clustering is a statistical tool and in some cases there is not a clear correlation among the lemmas of a cluster. In most of those cases, the clusters contain more than 10 elements, and we can discover patterns among subsets of those clusters, but it is difficult to find a common pattern followed by all lemmas.

The above results show that ML4PG can be useful to detect patterns in early stages of a development. Namely, it can be used to find relations among functions and their lemmas, common strategies followed in a library and fundamental lemmas applied in several proofs. Besides, if a user knows a library (e.g. the library defining natural numbers), ML4PG can show similarities between lemmas about natural numbers and lists, facilitating the use of the new library based on the previous knowledge of the user.

4. User scenario 2. ML4PG for detecting irrelevant libraries

An (abstract) sequential game can be represented as a tree with pay-off functions in the leaves, dictating the win or loss of each player when the game finishes there. Each internal node is owned by a player and a play of a game is a path from the root to a leaf. A *strategy* is a game where each internal node has chosen a child. A *Nash equilibrium* is a strategy in which no agent can change one or more of his choices to obtain a better overall result for himself. A strategy is a *subgame perfect equilibrium* if it represents/have a Nash equilibrium of every subgame of the original game.

In this scenario, we use ML4PG to analyse two Coq libraries that formalise that all sequential games have Nash equilibrium in binary games (games where each internal node has two choices) [33] and in the general case [32]. Note that unlike the other benchmarks presented throughout the paper, the files presented here are developed using plain Coq instead of SSReflect. ML4PG adapts to this change automatically.

It would be natural to assume that the proofs involved in verification of the two results will be very similar, and thus one could potentially hope for proof-pattern re-use. However, close inspection of these libraries can reveal that the actual proof strategies used in both libraries are different. Without ML4PG, such "negative" discovery would require user's time and experience in comparing Coq proofs. We instead give it as a challenge to ML4PG that takes only a few seconds to

BI_Exists	NashEq_Exists
Theorem BI_Exists : forall g, exists s, BI s /\ g = s2g s. Proof. deskolem_apply BI_fctExists. Qed. Theorem BI_fctExists : exists F, forall g, BI (F g)/\ g = s2g (F g). Proof. exists compBI. intro g. split. exact (compBI_is_BI g). exact (s2g_inv_compBI g). Qed.	Theorem NashEq_Exists : forall g, exists s, NashEq s /\ g = s2g s. Proof. deskolem_apply NashEq_fctExists. Qed. Theorem NashEq_fctExists : exists F, forall g, NashEq (F g)/\ g = s2g (F g). Proof. exists compBI. intro g. split. apply BI_is_NashEq. exact (compBI_is_BI g). exact (s2g_inv_compBI g). Qed.

TABLE 1. **Proofs of theorems `BI_Exists` and `NashEq_Exists`,** *coming from one library [33]; and grouped together by ML4PG.*

analyse the libraries. ML4PG loads the Coq files developed in [32,33] and a library about topological sorting [31] used in [32]. These Coq files include 145 theorems, and we choose the K-means algorithm and the value 5 as the granularity parameter to obtain clusters using ML4PG. ML4PG finds 32 clusters using those parameters, and their mean size is three elements per cluster. The question is: how can the user interpret these results, when he sees those 32 sets of approximately three lemmas/theorems on the Proof General screen?

It can be easily seen from ML4PG annotation of results that 21 of the 32 clusters (65%) are homogeneous clusters, thus the similarity between the proofs within one library is higher than across libraries. Starting first with those homogeneous clusters, we notice that

- 8 clusters (38%) contain lemmas about related functions, – as they use similar lemmas in their proofs.

  Example 4.1. As an example of this kind of clusters, ML4PG discovers a cluster with two lemmas from [33]: the first one (`BI_Exists`) states that for every game, there exists a strategy that makes the game to have Backward-Induction equilibrium (each player plays optimally at every node); the second lemma (`NashEq_Exists`) states the analogous result for the Nash equilibrium. See Table 1 for the proof of these two theorems.

- 6 clusters (28%) consist of lemmas about a concrete function.

  Example 4.2. In [32], there is a function called `StratPref` that given an agent and two strategies decides which is the best one. ML4PG finds a cluster with two lemmas: the first one (`StratPref_dec`) states the decidability of the function; and the second one states that the function produces an irreflexive relation.

- 4 clusters (19%) contain theorems that use other theorems of the cluster in their proofs.

Binary case	General case

`Lemma SGP_is_NashEq :`
`forall s : Strategy, SGP s -> NashEq s.`
`Proof.`
`induction s.`
`unfold NashEq. intros _. induction s'.`
`intros. unfold stratPO. unfold agentConv in H.`
`rewrite (H a). trivial.`
`unfold agentConv. intros. contradiction.`
`unfold SGP. intros [_ [_ done]]. trivial.`
`Qed.`
	`Lemma SPE_is_Eq :`
	`forall s : Strat, SPE s -> Eq s.`
	`Proof.`
	`intros. destruct s; simpl in H; tauto.`
	`Qed.`

TABLE 2. **Proof of the theorem stating that Subgame Perfect Equilibrium implies Nash Equilibrium. Left.** *Binary case.* **Right.** *General case. The lemma statements are very similar; however, the structure of the proofs is completely different; hence, ML4PG does not group these proofs together.*

This quick analysis would show that some obvious grouping of proofs within one library was made by ML4PG. But, unless we are interested in a particular proof technique appearing in one of them, we direct our attention to patterns found across the libraries, hoping to find some common proof methods across the developments.

In the case of heterogeneous clusters, all the clusters consist of lemmas about auxiliary functions (for instance, about different properties of lists) that are common in all the libraries we are studying. However, there is no correlation among the important theorems of these libraries; see Table 2. Even if [32] is a generalisation of the work presented in [33], the proofs for Nash equilibrium are completely different, mainly for two reasons. First, the datastructures that are used in each development are too different, and therefore the lemmas about them do not have a strong correlation. In addition, the approaches that are used to prove them are completely different: one based on a procedure called backward induction [33] and the other is based on the fact that the preference of players is acyclic.

The results do not vary much when we try changing the clustering algorithm and the granularity values – reducing the granularity value produces bigger homogeneous clusters, but has little effect on heterogeneous clusters. As can be seen from this example, the ML4PG-based proof-pattern check could be an easy and fast way of getting the information about the absence of recyclable patterns across the libraries.

Note that in this case study, the total number of lemmas is smaller than in the previous case-study (145 theorems); but the feature extraction mechanism of ML4PG automatically adapts to this, and handles statistics of small data sets as well as statistics of bigger data sets.

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

```
0  :  iconst 1
1  :  istore 1
2  :  iload 0
3  :  ifeq 13
4  :  iload 1
5  :  iload 0
6  :  imul
7  :  istore 1
8  :  iload 0
9  :  iconst 1
10 :  isub
11 :  istore 0
12 :  goto 2
13 :  iload 1
14 :  ireturn
```

```
Fixpoint helper_fact (n a) :=
match n with
| 0 => a
| S p => helper_fact p (n * a)
end.

Definition fn_fact (n : nat) :=
     helper_fact n 1.
```

FIGURE 1. **Factorial function**. **Left:** *Java program for computing the factorial of natural numbers.* **Centre:** *Java bytecode associated with the Java program.* **Right:** *tail recursive version of the factorial function in Coq/SSReflect.*

5. User scenario 3. A team-based development

In the last scenario, we turn to team-based applications of Coq and ML4PG. For this purpose, we translate the ACL2 proofs of correctness of the Java Virtual Machine (JVM) [23] into Coq/SSReflect. JVM [28] is a stack-based abstract machine which can execute Java bytecode. We have modelled an interpreter for JVM programs in Coq/SSReflect. From now on, we refer to our machine as "SJVM" (for SSReflect JVM).

An industrial scenario of interactive theorem proving may involve distribution of work-load across a team, and a bigger proportion of routine or repetitive cases. Here, the inefficiency often arises when programmers use different notation to accomplish very similar tasks, and thus a lot of work gets duplicated, see also [11]. We tested ML4PG in exactly such a scenario: we assumed that a programming team is collectively developing proofs of *the soundness of the specification, and the correctness of the implementation* of Java bytecode for a dozen of programs computing multiplication, powers, exponentiation, and other functions about natural numbers. A new team member then tries to learn the important proof patterns while trying to prove similar results for a new function – factorial.

Given a specific Java method, we can translate it to Java bytecode using a tool such as `javac` of Sun Microsystems. Such a bytecode can be executed in SJVM provided a schedule (a list of thread identifiers indicating the order in which the threads are to be stepped), and the result will be the state of the JVM at the end of the schedule. Moreover, we can prove theorems about the SJVM model behaviour when interpreting that bytecode.

Example 5.1. The bytecode associated with the factorial program can be seen in Figure 1.

The state of the SJVM consists of 4 fields: a *program counter* (a natural number), a set of registers called *locals* (implemented as a list of natural numbers), an operand *stack* (a list of natural numbers), and the bytecode *program* of the method being evaluated.

Java bytecode, like the one presented in Figure 1, can be executed within SJVM. However, more interestingly than merely executing Java bytecode, we can prove the correctness of the implementation of the Java bytecode programs using Coq/SSReflect. For instance, in the case of the factorial program, the new team member is asked to prove the following theorem, which states the correctness of the `factorial` bytecode.

Theorem 5.2. *Given a natural number n and the factorial program with n as an input, SJVM produces a state which contains $n!$ on top of the stack running the bytecode associated with the program.*

The proof of theorems like the one above always follows the same methodology adapted from ACL2 proofs about Java Virtual Machines [23] and which consists of the following three steps.

(1) Write in Coq/SSReflect the specification of the function and the algorithm, and prove that the algorithm satisfies the specification.
(2) Write the JVM program within Coq/SSReflect, define the function that schedules the program (this function will make SJVM run the program to completion as a function of the input to the program), and prove that the resulting code implements this algorithm.
(3) Prove total correctness of the Java bytecode.

Using this methodology, we have proven the correctness of several programs related to arithmetic (multiplication of natural numbers, exponentiation of natural numbers, and so on); see [20]. The proof of each theorem was done independently from others to model a distributed proof development.

Therefore, we simulate the following scenario. Suppose a new developer tackles for the first time the proof of Theorem 5.2, and he knows the general methodology to prove it and has access to the library of programs previously proven by other users. In this situation the different notation employed by different users obscures some common features. ML4PG would be a good alternative to the manual search for proof patterns.

Let us focus on the first step of the methodology – that is, the proof of the equivalence between the specification of the factorial function (which is already defined in SSReflect using the function `factorial` having the notation `n'!` for `factorial n`) and the algorithm, see Table 3. The Java factorial function is an iterative function; and the algorithm is written in Coq as a tail recursive function, see the right side of Figure 1. In the available SJVM libraries, all the tail recursive functions are defined using an auxiliary function, called the *helper*, and a wrapper for such a function. Discovering this fact is the first challenge for ML4PG. Suppose the new team member has stopped after one proof-step of trying to prove

Factorial	Exponentiation

`Lemma fn_fact_is_theta n : fn_fact n = n‘!.`
`Proof.`
`rewrite /fn_fact.`
`by rewrite helper_fact_is_theta mul1n.`
`Qed.`
 	
`Lemma helper_fact_is_theta n a :`
`helper_fact n a = a * n‘!.`
`Proof.`
`move : n a; elim : m => [a m| m IH n a /=].`
`by rewrite /theta_fact fact0 muln1.`
`by rewrite IH /theta_fact factS`
`mulnA [a * _]mulnC.`
`Qed.`
	`Lemma fn_expt_is_theta n m : fn_expt n m = n^m.`
	`Proof.`
	`by rewrite /fn_expt helper_expt_is_theta`
	`muln.`
	`Qed.`
	
	`Lemma helper_expt_is_theta n m a :`
	`helper_expt n m a = a * (n ^ m).`
	`Proof.`
	`move : a; elim : n => [a| n IH a /=].`
	`by rewrite /theta_expt expn0 muln1.`
	`by rewrite IH /theta_expt expnS`
	`mulnA [a * _]mulnC.`
	`Qed.`

TABLE 3. **Proofs of equivalence of the tail-recursive and recursive versions of functions exponentiation and factorial, following Proof Strategy 5.3.** *The left-hand-side shows a few initial proof steps for* `fn_fact_is_theta`*, leading to a deadlock. The right-hand-side shows the lemma (*`fn_expt_is_theta`*) suggested by ML4PG (see Table 4) and an auxiliary lemma used to prove it. In italics is the proof reconstruction by analogy.*

the lemma `fn_fact_is_theta` in a naive way, without a helper function; see Table 3. He cannot proceed, and calls ML4PG for a hint. The suggestions provided by ML4PG in this case are the proofs of step (1) for three iterative programs: the multiplication, the exponentiation and the power of natural numbers; see e.g. Lemma `fn_expt_is_theta` in Table 3. It is easy to notice that all of them use an auxiliary lemma (like `helper_expt_is_theta`), and thus follow the same proof strategy:

Proof Strategy 5.3. *Prove an auxiliary lemma about the helper considering the most general case. For example, if the helper function is defined with formal parameters n, m, and a, and the wrapper calls the helper initializing a at* 0*, the helper theorem must be about* `(helper n m a)`*, not just about the special case* `(helper n m 0)`*. Subsequently, instantiate the lemma for the concrete case.*

The technical details are as follows. ML4PG correctly suggested similar lemmas to lemma `fn_fact_is_theta`. Table 4 shows the results for different choices of algorithms and parameters, and we highlight the most precise and helpful ML4PG result. In case the user is unsure of the optimal machine-learning parameters, he could use a “top-down approach”. The highest granularity level does not produce any result. But, if we decrease the granularity level to 4, ML4PG spots some interesting similarities using the K-means algorithm. If this is not enough to discover Proof Strategy 5.3, one can decrease the granularity level to 3, for which ML4PG discovers four lemmas following the same general scheme.

Algorithm:	$g = 1$ ($n = 16$)	$g = 2$ ($n = 18$)	$g = 3$ ($n = 21$)	$g = 4$ ($n = 24$)	$g = 5$ ($n = 29$)
K-means	$30^{a,b,d}$	$\mathbf{4}^{a-d}$	$\mathbf{4}^{a-d}$	$2^{c,d}$	0
E.M.	21^{a-d}	7^{a-d}	7^{a-d}	0	0
FarthestFirst	28^{a-d}	25^{a-d}	0	0	0

TABLE 4. **A series of clustering experiments discovering Proof Strategy 5.3.** *The table shows the sizes of clusters containing: a) Lemma about JVM multiplication program, b) Lemma about JVM power program, c) Lemma about JVM exponentiation program, and d) Lemma about JVM factorial program. The size of the data set is 147 lemmas, in bold is the cluster that finds exactly the four benchmark examples. Again, the lemmas grouped by clusters are consistently found for various algorithms and granularity values; and the K-means algorithm provides the most accurate clusters using* 3 *as granularity value.*

On the basis of these suggestions, the new team member can try to reconstruct the missing auxiliary lemma and the missing proof steps in the main lemma by analogy. Table 3 shows such analogical reconstruction in italics. This takes him through the first step of the general proof scheme.

In the second stage, he needs to prove that the Java bytecode implements the factorial algorithm. Again, after a few proof-steps, the user gets stuck and cannot continue the proof, see Table 5. If ML4PG is invoked at this point, it suggests three lemmas (using K-means algorithm and 3 as granularity value) that are used to prove that the three Java bytecode programs implement multiplication, exponentiation and power algorithms, respectively. All these Java bytecode programs are iterative and involve a loop, and it is easy to notice that the proofs follow the same proof strategy:

Proof Strategy 5.4. *Prove that the loop implements the helper using an auxiliary lemma. Such a lemma about the loop must consider the general case as in the case of Proof Strategy 5.3. Subsequently, instantiate the result to the concrete case.*

Using this strategy and by analogy with the proofs of the other lemmas of the cluster, the user can finish the proof of lemma `program_is_fn_fact`, Table 5 shows in italics the reconstruction of that proof.

Finally, it remains to prove the total correctness of the Java bytecode (Theorem 5.2). ML4PG finds that all the proofs of the total correctness of the different programs are similar and follow the same proof pattern which consists of applying the lemmas obtained from steps (1) and (2), see Table 6. Again, Table 6 illustrates the scenario of calling ML4PG on demand, and using its suggestions to reconstruct the proof by analogy. Following these guidelines, Theorem 5.2 can be formalised in Coq/SSReflect by analogy with a similar lemma for e.g. exponentiation, obtaining as a result the proof of the correctness of the factorial Java bytecode, as shown in Table 6; see also [20] for the full proof.

Factorial

```
Lemma program_is_fn_fact n :
run (sched_fact n)(make_state 0 [::n] [::] pi_fact)=
(make_state 14 [::0;fn_fact n ] (push (fn_fact n )[::])pi_fact).
Proof.
rewrite run_app.
rewrite loop_is_helper_fact.
Qed.

Lemma loop_is_helper_fact n a :
run (loop_sched_fact n)(make_state 2 [::n;a] [::] pi_fact)=
(make_state 14 [::0;(helper_fact n a)] (push (helper_fact n a)[::])pi_fact)
Proof.
move : a; elim : n => [// | n IH a].
by rewrite -IH subn1 -pred_Sn [_ * a]mulnC.
Qed.
```

Exponentiation

```
Lemma program_is_fn_expt n m :
run (sched_expt n m)(make_state 0 [::n;m] [::] pi_expt)=
(make_state 14 [::0;fn_expt n m] (push (fn_expt n m)[::])pi_expt).
Proof.
rewrite run_app loop_is_helper_expt.
Qed.

Lemma loop_is_helper_expt n m a :
run (loop_sched_expt n)(make_state 2 [::n;m;a] [::] pi_expt)=
(make_state 14 [::0;(helper_expt n m a)] (push (helper_expt n m a)[::])pi_expt)
Proof.
move : n a; elim : m => [// | m IH n a].
by rewrite -IH subn1 -pred_Sn.
Qed.
```

TABLE 5. **Proofs that the Java bytecodes implement the factorial and exponentiation algorithms.** *When the user tries to prove* `program_is_fn_fact`*, he stops after one proof step (top table) and calls ML4PG. ML4PG suggests a few theorems, like e.g.* `program_is_fn_expt` *(bottom table). It would work for e.g. K-means algorithm and granularity values from 1 to 4, but using 4 as granularity value the cluster only contains these two lemmas. In italics, the user reconstructs the proof by analogy with* `program_is_fn_expt` *following Proof Strategy 5.4.*

The clusters found in the JVM scenario are heterogeneous since they belong to different libraries. The clusters obtained for the different steps are in the category of clusters that consists of lemmas with the same proof structure and that use analogous lemmas. This is an interesting kind of clusters since the analogous lemmas could be automatically generated using techniques presented in [22].

Factorial

```
Theorem total_correctness_fact n sf :
sf = run (sched_fact n)(make_state 0 [::n] [::] pi_fact)->
next_inst sf = (HALT,0\%Z)/\ top (stack sf)= (n'!).
Proof.
move => H; split
 ; rewrite H program_is_fn_fact fn_fact_is_theta.
Qed.
```

Exponentiation

```
Theorem total_correctness_expt n m sf :
sf = run (sched_expt m)(make_state 0 [::n;m] [::] pi_expt)->
next_inst sf = (HALT,0%Z)/\ top (stack sf)= (n^m).
Proof.
 by move => H; split; rewrite H program_is_fn_expt fn_expt_is_theta.
Qed.
```

TABLE 6. **Proofs of total correctness for exponentiation and factorial programs, cf. Theorem 5.2**. *The top table shows the initial step to prove Theorem 5.2 (`total_correctness_fact`). ML4PG suggests a few theorems, like e.g. `total_correctness_expt` (see the bottom table). ML4PG provides this suggestions for different parameters (e.g. K-means algorithm and granularity values from 1 to 4), but using 4 as granularity value and K-means as algorithm the cluster only contains these two lemmas. In italics, the user reconstructs the proof by analogy with the theorem `total_correctness_expt`.*

6. Conclusions and Future Work

We have presented three scenarios, of very different nature and domain, to test the capabilities of statistical proof-pattern recognition. We have observed that ML4PG's feature extraction provides sufficiently robust results, tested using a few most common clustering algorithms (cf. Table 4). Judging by the experiments, K-means algorithm is the most reliable algorithm, showing very stable results. The best value for granularity depends on the size of the library, in big libraries (cf. User Scenario 1) granularity values 4 and 5 return the most accurate clusters; however, in small libraries (cf. User Scenarios 2 and 3) the granularity value of 3 produces better results. ML4PG in general requires minimum user effort – mainly concerning adjustments of the granularity parameter to obtain the result of the required precision. ML4PG is very fast and gives instant outputs allowing the user to have quick search/evaluation in the interactive manner.

The most valuable feature of ML4PG is that it works equally well with any library we tried; irrespective of the subject domain or the size of the libraries. This property can be used to find patterns across subjects, libraries, and users; – as our case studies illustrate. Moreover, ML4PG discovers two kinds of clusters:

homogeneous (all the lemmas of the cluster belong to the same library) and heterogeneous (the lemmas of the cluster belong to different libraries). Most of the time, the relation among the elements of a homogeneous cluster is clear (same proof structure, same lemmas or analogous lemmas). On the contrary, the relation among the elements of a heterogeneous cluster is more subtle (e.g. a general proof strategy or the use of some kind of auxiliary lemma).

Work is under way to incorporate the following extensions into ML4PG (see [20] for the most recent ML4PG versions):

- a more sophisticated proof-patch identification for bigger proofs. This paper is based on an ML4PG version that uses only patches of the first few steps in a proof, but see [20] for the experimental version that uses proof-patches to cover entire proofs.
- have a robust data-mining of type declarations and (co-)inductive definitions, alongside the currently used proof-analysis.
- introducing a recurrent approach to feature-extraction similar to [22].

A longer-term project is to generate auxiliary lemmas and definitions automatically, on the basis of statistically discovered patterns. We have already done that for ACL2 [22]; however, extrapolation of the techniques of [22] from first-order untyped language to higher-order dependently-typed language is a difficult task.

Acknowledgments

We would like to thank Marco Gaboardi and Vladimir Komendantsky for proof-reading the paper; their suggestions helped us to improve presentation.

References

[1] A. Amorim, N. Collins, A. DeHon, D. Demange, C. Hricu, D. Pichardie, B. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[2] A. Asperti, F. Guidi, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. A Content Based Mathematical Search Engine: Whelp. In *Post-Proceedings of the TYPES'04 International Conference*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32, 2006.

[3] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. The Matita interactive Theorem prover. In *23rd International Conference on Automated Deduction (CADE'11)*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69, 2011.

[4] D. Aspinall. Proof General: A Generic Tool for Proof Development. In *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–43, 2000.

[5] D. Basin, A. Bundy, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[6] N. Benton. Machine Obstructed Proof: How many months can it take to verify 30 assembly instructions?, 2006.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Constructions*. Springer-Verlag, 2004.

[8] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] A. Blum. Learning boolean functions in an infinite attribute space. *Machine Learning*, 9(4):373–386, 1992.

[10] A. Bove, P. Dybjer, and U. Norell. A Brief Overview of Agda — A Functional Language with Dependent Types. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78, 2009.

[11] A. Bundy, D. Hutter, C. Jones, and J S. Moore. AI meets Formal Software Development (Dagstuhl Seminar 12271). *Dagstuhl Reports*, 2(7):1–29, 2012.

[12] Coq development team. The Coq Proof Assistant Reference Manual, version 8.4. Technical report, 2012.

[13] H. Duncan. *The use of Data-Mining for the Automatic Formation of Tactics*. PhD thesis, University of Edinburgh, 2002.

[14] G. Gonthier. Formal proof - the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[15] G. Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In *4th Conference on Interactive Theorem Proving (ITP'13)*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179, 2013.

[16] G. Gonthier and A. Mahboubi. An introduction to small scale reflection. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[17] T. Hales. The flyspeck project fact sheet. Project description available at `http://code.google.com/p/flyspeck/`, 2005.

[18] M. Hall et al. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[19] J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology within Coq/SSReflect. *To be published in Transactions on Computational Logic*, 2013.

[20] J. Heras and E. Komendantskaya. ML4PG: downloadable programs, manual, examples, 2012–2013. `http://staff.computing.dundee.ac.uk/katya/ML4PG/`.

[21] J. Heras and E. Komendantskaya. Statistical proof pattern recognition: Automated or interactive? In *Proceedings of the 20th Automated Reasoning Workshop (ARW'13)*, pages 25–26, 2013.

[22] J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *19th Logic for Programming Artificial Intelligence and Reasoning (LPAR-19)*, volume 8312 of *Lecture Notes in Computer Science*, pages 389–406, 2013.

[23] J S. Moore. *Models, Algebras and Logic of Engineering Software*, chapter Proving Theorems about Java and the JVM with ACL2, pages 227–290. IOS Press, 2004.

[24] M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.

[25] E. Komendantskaya, J. Heras, and G. Grov. Machine Learning for Proof General: interfacing interfaces. *Electronic Proceedings in Theoretical Computer Science*, 118:15–41, 2013.

[26] R. Krebbers and B. Spitters. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science*, 9(1):1–27, 2013.

[27] D. Kühlwein et al. Overview and evaluation of premise selection techniques for large theory mathematics. In *6th International Joint Conference on Automated Reasoning (IJCAR'12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 378–392, 2012.

[28] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification: Java SE 7 Edition, 2012.

[29] MATLAB. *version 7.14.0 (R2012a)*. The MathWorks Inc., Natick, Massachusetts, 2012.

[30] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[31] S. Le Roux. Acyclicity and finite linear extendability: a formal and constructive equivalence. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, Emerging Trends Proceedings, pages 154–169, 2007.

[32] S. Le Roux. 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)*, volume 5674 of *Lecture Notes in Computer Science*, pages 293–309, 2009.

[33] R. Vestergaard. A constructive approach to sequential nash equilibria. *Information Processing Letter*, 97:46–51, 2006.

Jónathan Heras
School of Computing, University of Dundee, UK
e-mail: `jonathanheras@computing.dundee.ac.uk`

Ekaterina Komendantskaya
School of Computing, University of Dundee, UK
e-mail: `katya@computing.dundee.ac.uk`