Proof-Pattern Recognition and Lemma Discovery in ACL2[*]

Jónathan Heras[1], Ekaterina Komendantskaya[1], Moa Johansson[2], and Ewen Maclean[3]

[1] School of Computing, University of Dundee, UK
[2] Dept. of Computer Science and Engineering, Chalmers University, Sweden
[3] School of Informatics, University of Edinburgh, UK
{jonathanheras,katya}@computing.dundee.ac.uk, moa.johansson@chalmers.se,
E.Maclean@ed.ac.uk

Abstract. We present a novel technique for combining statistical machine learning for proof-pattern recognition with symbolic methods for lemma discovery. The resulting tool, **ACL2(ml)**, gathers proof statistics and uses statistical pattern-recognition to pre-processes data from libraries, and then suggests auxiliary lemmas in new proofs by analogy with already seen examples. This paper presents the implementation of ACL2(ml) alongside theoretical descriptions of the proof-pattern recognition and lemma discovery methods involved in it.
Keywords: *Theorem Proving, Statistical Machine-Learning, Pattern Recognition, Lemma Discovery, Analogy.*

1 Introduction

Over the last few decades, theorem proving has seen major developments. *Automated (first-order) theorem provers (ATPs)* (e.g. E, Vampire, SPASS) and SAT/SMT solvers (e.g. CVC3, Yices, Z3) are becoming increasingly fast and efficient. *Interactive (higher-order) theorem provers (ITPs)* (e.g. Coq, Isabelle/HOL, Agda, Mizar) have been enriched with dependent types, (co)inductive types, type classes and provide rich programming environments.

The main conceptual difference between ATPs and ITPs lies in the styles of proof development. For ATPs, the proof process is primarily an automatically performed *proof search* in a *first-order* language. In ITPs, the proof is guided by *the user* who specifies which tactics to apply. ITPs often work with *higher-order* logic and type theory.

Communities working on development, implementation and applications of ATPs and ITPs have accumulated big corpora of electronic proof libraries. However, the size of the libraries, as well as their technical and notational sophistication often stand in the way of efficient knowledge re-use. Very often, it is easier to start a new library from scratch rather than search the existing proof libraries for potentially common heuristics and techniques.

Pattern-recognition [2] is an area of machine-learning that develops statistical methods for discovering patterns in data. In the statistical sense, a pattern is a correlation of several numeric features, by which the data is represented. For instance, the correlation between the theorem statement and the auxiliary lemmas required in its proof can be data-mined to improve automation of premise selection in ATPs [7, 8, 18, 22, 23]. The history of successful and unsuccessful proof attempts and proof steps can also be used to inform interactive proof development in ITPs [9, 17].

Statistical machine-learning methods are well-suited for fast processing of big proof libraries, adapt well to proofs of varied sizes and complexities (first- or higher-order); and are generally tolerant to noise. However, they have very weak capacities for conceptualisation. For instance, ML4PG [17] only displays the families of related proofs to the Coq user, and can tell the correlation of which features formed the pattern, but it neither explains why this happens nor formulates any conceptual proof hints.

While statistical methods focus on extracting information from existing large theory libraries, many symbolic methods are instead concerned with automating the discovery of lemmas in new theories [4, 6, 11, 13, 20, 21], while relying on existing proof strategies, e.g. proof-planning and rippling [1]. These systems are naturally more deterministic and algorithmic than statistical AI.

IsaCoSy [13] and IsaScheme [21] are two term synthesis systems built on top of Isabelle. They generate candidate conjectures which are filtered through a counter-example checker. Surviving conjectures are passed to an automated prover and those proven are added to the theory. The systems differ in their heuristics for term generation: IsaCoSy only generates irreducible terms, while IsaScheme uses *schemes* [3] to specify the shapes of candidate theorems. A similar system is QuickSpec and its extension HipSpec [4, 5], which generates equational theorems about Haskell programs, using congruence closure for conjecture generation. MATHsAiD [20] generates theories by forward reasoning from a given set of axioms, applying various 'interestingness' heuristics.

Symbolic methods have limits: they can be slow on large inputs due to the increase in the search space, rely on having access to good counter-example finders for filtering of candidate conjectures and require existing proof strategies for proving those remaining.

In this paper, we show that it is possible to combine statistical and symbolic methods to get the best of both worlds: statistical pattern-recognition methods are well-suited for finding families of similar proofs; symbolic tools can use this data for more efficient lemma discovery. Feeding outputs of one algorithm to the other leads to a very natural proof-pattern recognition system, see Figure 1. To realise this general thesis about the synthesis of the two styles of proof-pattern recognition, we have made the following methodological choices:

1) We chose the ACL2 prover for our experiments: it is based on first-order logic and has features of both ITPs and ATPs. ACL2 will try to prove given conjectures automatically, but strongly relies on the user having supplied the auxiliary lemmas required for rewriting. Thus, the user must often intervene
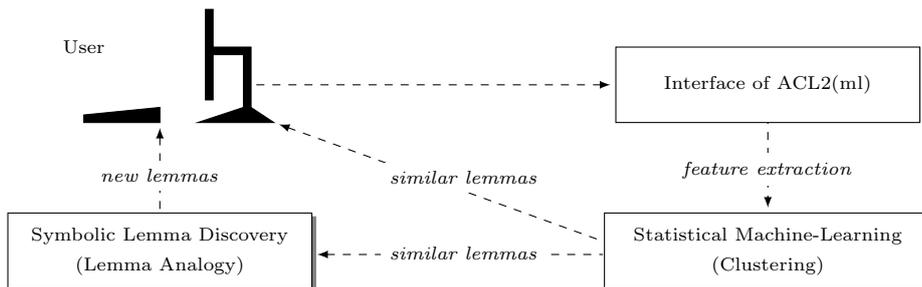
Fig. 1. *Architecture of proof-pattern recognition methods in ACL2(ml).* The Emacs interface for ACL2 extracts important features from ACL2 theorems, connects to machine-learning software, clusters the theorems there, and sends the result (families of similar theorems) to the screen. In addition, for an unproved theorem T, it sends the cluster C of T to the Lemma Analogy tool, which in its turn generates an auxiliary lemma by analogy with auxiliary lemmas of the theorems in C.

and advance the proof by adding new lemmas, much like in ITPs. Section 2 gives a brief introduction to ACL2.

2) On the machine-learning side, we chose to adapt the ML4PG [17] design of *interactive* proof-pattern recognition; cf. Figure 1. We extended this work with a new feature extraction algorithm for ACL2(ml), based on term trees and recurrent clustering. Much like in ML4PG, we discover families of related lemmas, rather than premise hierarchies like in [18, 22, 23]. Unlike [18, 22, 23], we use clustering instead of supervised learning; and do not use sparse methods. Section 3 explains our new method in detail.

3) On the symbolic side, we develop a novel lemma generation approach which uses the statistical suggestions to reduce the search space. Speculating the correct lemma in a proof can often be the *eureka* moment which allows a proof to succeed. It is these eureka lemmas which we would like to reproduce automatically. We propose an analogy driven approach where the term structure of an example *source lemma* is analysed to produce an analogous *target lemma* for a given target problem. In the context of ITP, the user is then presented with an auxiliary lemma suggestion which is customised to the particular problem on which he is working. Section 4 is devoted to this subject.

We implement the above choices in the ACL2 extension ACL2(ml), available at [10]. Finally, we conclude with evaluation of this method in Section 5. The results we obtain with ACL2 have a wider significance, as methodology we develop here would apply to a wide range of first-order theorem provers.

2   Background

ACL2 [14, 16] (standing for *A Computational Logic for an Applicative Common Lisp*) is a programming language, a logic, and a theorem prover supporting reasoning in the logic. The ACL2 programming language is an extension of an applicative subset of Common Lisp. The ACL2 logic is an untyped first-order

```
;; Factorial
1 (defun fact (n) (if (zp n) 1 (* n (fact (- n 1)))))

2 (defun helper-fact (n a) (if (zp n) a (helper-fact (- n 1) (* a n))))

3 (defun fact-tail (n) (helper-fact n 1))

4 (defthm fact-fact-tail (implies (natp n) (equal (fact-tail n) (fact n))))

;; 2^n
1 (defun power (n) (if (zp n) 1 (* 2 (power (- n 1)))))

2 (defun helper-power (n a) (if (zp n) a (helper-power (- n 1) (+ a a))))

3 (defun power-tail (n) (helper-power n 1))

4 (defthm power-power-tail (implies (natp n) (equal (power-tail n) (power n))))

;; Fibonacci
1 (defun fib (n) (if (zp n) 0 (if (equal n 1) 1 (+ (fib (- n 1)) (fib (- n 2))))))

2 (defun helper-fib (n j k) (if (zp n) j (if (equal n 1) k (helper-fib (- n 1) k (+ j k)))))

3 (defun fib-tail (n) (helper-fib n 0 1))

4 (defthm fib-fib-tail (implies (natp n) (equal (fib-tail n) (fib n))))
```

Fig. 2. *ACL2 definitions and theorems.* 1: recursive arithmetic functions. 2: helpers of tail-recursive arithmetic functions. 3: tail-recursive arithmetic functions. 4: Equivalence theorems of recursive and tail-recursive functions.

logic with equality, used for specifying properties and reasoning about the functions defined in the programming language. All the variables in the formulas allowed by the ACL2 system are implicitly universally quantified. The syntax of its terms and formulas is that of Common Lisp.

Example 1. Given the recursive and tail-recursive functions to compute factorial, 2^n and Fibonacci, the ACL2 user can specify the equivalence between the functions as shown in Figure 2 with theorems `fact-fact-tail`, `power-power-tail` and `fib-fib-tail`. Note the similarity between these lemmas.

ACL2 has both automatic and interactive features. It is automatic in the sense that once a `defthm` command is submitted, the user can no longer interact with the system (the ACL2 proof engine applies a collection of automatic tactics until either the conjecture is proven or none of the tactics is applicable). Often, non-trivial results cannot be proven on the first attempt. The user then has to interact with the prover by supplying a suitable collection of definitions and auxiliary lemmas, used in subsequent proofs as rewriting rules. These lemmas are suggested by a preconceived hand proof or by inspection of failed proofs; this kind of interaction is referred to as *The Method* [14].

Example 2. ACL2's first attempt at proving the conjecture `fact-fact-tail` of Example 1 fails, and the user needs to introduce a lemma (inspired by the failed proof of `fact-fact-tail`), which is automatically proven by ACL2:

```
(defthm fact-fact-tail-helper (implies (and (natp n) (natp a))
       (equal (helper-fact n a) (* a (fact n)))))
```

ACL2 can now automatically prove the conjecture `fact-fact-tail`, using this auxiliary result. In later sections, we will show how ACL2(ml) automatically detects similar theorems and generates auxiliary lemmas for them.

The proofs of equivalence between the recursive and tail-recursive functions in Figure 2 follow a common pattern: the equivalence theorems are not proven by ACL2 in the first attempt, and the user must introduce auxiliary lemmas about the `helper` functions. In general, the detection of common patterns and the generation of new lemmas from those patterns is based on user's experience and can be challenging. The ACL2 distribution consists of several libraries containing hundreds of theorems, developed by several users with their own notations. It can be a challenge to detect patterns across different users, notations and libraries. Moreover, given two similar theorems T_1 and T_2, the lemmas used to prove T_1 can be substantially different to the ones needed for T_2 (e.g. different function symbols, different lemma structures, additional conditions, new concepts).

In the next sections, we describe how these challenges can be addressed automatically. We consider two running examples. The first one is the JVM library developed in [12]. This library contains the correctness proofs of the Java bytecode associated with several arithmetic programs such as multiplication, factorial and Fibonacci, including the theorems of Figure 2. As a second running example, we consider the *Lists* library presented in [15]. From this library, we consider three functions that will appear later in the paper: `sort` (that sorts a list of natural numbers), `rev` (which reverses a list), and `int` (which takes two lists as arguments and returns the list of elements that appear in both); and three theorems about these functions.

```
(defthm sortsort (implies (nat-listp x) (equal (sort (sort x)) (sort x))))
(defthm revrev (implies (true-listp x) (equal (rev (rev x)) x)))
(defthm int-x-x (implies (true-listp x) (equal (int x x) x)))
```

3 Statistical Proof-Pattern Recognition with ACL2(ml)

We present a statistical proof-pattern recognition extension to ACL2, called **ACL2(ml)**. Its implementation design follows the ML4PG tool for Coq [17]: this Emacs-based machine-learning extension works on the background of the theorem prover, gathers statistics of proof features, and then, on user's request, connects to a machine-learning toolbox (MATLAB or Weka); groups the proofs using *clustering* algorithms, and displays families of related proofs to the user. This approach allows real-time interaction between the user, the prover, and the machine-learning systems; as [17] explains in detail.

Here, we concentrate only on original components of ACL2(ml) as compared to ML4PG. There are two such features: a novel algorithm for term tree feature extraction and the method of *recurrent clustering*.

	impl	natp	equal	fact-t.	fact	n	n'	n"
impl	0	1	1	0	0	0	0	0
natp	0	0	0	0	0	1	0	0
equal	0	0	0	1	1	0	0	0
fact-t.	0	0	0	0	0	0	1	0
fact	0	0	0	0	0	0	0	1
n	0	0	0	0	0	0	0	0
n'	0	0	0	0	0	0	0	0
n"	0	0	0	0	0	0	0	0
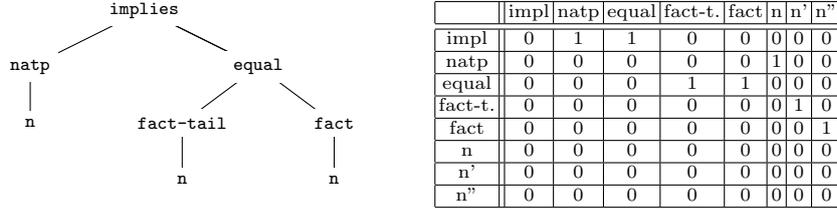
Fig. 3. *Term tree and adjacency matrix for Theorem* `fact-fact-tail`*; assuming the root-to-leaf direction of edges.*

Term tree feature **extraction in ACL2(ml).** The discovery of statistically significant features in data is a research area of its own in machine learning, known as *feature extraction* [2]. Irrespective of the particular algorithm used, most pattern-recognition tools will require that the features have numeric values; and the number of selected features is limited and fixed (sparse methods, like the ones applied in e.g. [18, 22, 23], are the exception to the latter rule). The feature extraction algorithm implemented in ML4PG was based on correlation between the goal shapes, the tactics and tactic parameters within a few steps of the interactive proof. As ACL2 is not a tactic-driven language, we need to extract features from the ACL2 terms directly. Feature extraction from terms or *term trees* is common to most feature-extraction algorithms implemented in automated theorem provers: see e.g. [18, 22, 23].

Definition 1 (Term tree). *A variable or a constant is represented by a tree consisting of one single node, labelled by the variable or the constant itself. A function application $f(t_1, \ldots, t_n)$ is represented by the tree with the root node labelled by f, and its immediate subtrees given by trees representing $t_1, \ldots, t_n$.*

Example 3. Theorem `fact-fact-tail` from Example 1 can be represented by the term tree of Figure 3.

A variety of methods exists to represent trees as matrices, for instance using *adjacency matrices* and *incidence matrices*, with the former shown in Figure 3. The adjacency matrix and the various previous methods of term tree feature extraction (e.g. [18, 22, 23]) share the following common properties: different tree nodes are represented by distinct matrix entries (*features*); the matrix entries (*feature values*) are binary; and the size of the matrix depend on the tree size. For big libraries, such matrices can 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 in [18, 22, 23].

In ACL2(ml), we implement a more compact feature extraction, where a total number of features is fixed for all libraries at 49, and the average density of matrices is 20%. The smaller and denser feature vectors can be used to data-mine both big and small data sets. Large and sparse feature vectors would require to use big data sets of proofs, where the size of the example data set is comparable to

	variables	arity 0	arity 1	arity 2
td0	0	0	0	[implies]
td1	0	0	[natp]	[equal]
td2	[n]	0	[fact-tail]::[fact]	0
td3	[n]::[n]	0	0	0

	variables	arity 0	arity 1	arity 2	arity 3
td0	0	0	0	0	[if]
td1	0	[1]	[zp]	[*]	0
td2	[n]::[n]	0	[fact]	0	0
td3	0	0	0	[-]	0
td4	[n]	[1]	0	0	0

Table 1. *Left. Dense term tree feature extraction matrix for theorem* fact-fact-tail. *Right. Dense term tree feature extraction matrix for definition* fact. *The operator "::" indicates the concatenation of values (e.g. 4::5=45).*

the number of features. In our interactive setting, it is crucial that the ACL2(ml) tool works equally well with both big and small proof libraries; and that the user can interact with it at any stage of the proof development.

We develop a method to overcome the problem of having to track a large (potentially unlimited) number of ACL2 symbols by a finite number of features, as follows. The ACL2 symbols are represented by distinct feature values given by rational numbers. The feature values are computed dynamically by *recurrent clustering* algorithm, thus reflecting the recursive nature of the ACL2 functions and proofs. The *features* are given by the finite number of properties common to all possible term trees: the term arity and the term tree depth; see Table 1. This is formalised in the following definitions.

Definition 2 (Term tree depth level). *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.

Definition 3 (ACL2(ml) term tree matrices). *Given a term tree T for a term with signature Σ, and a function $[.] : \Sigma \to \mathbb{Q}$, the ACL2(ml) term tree matrix M_T is a 7×7 matrix that satisfies the following conditions:*
  — the $(0, j)$-th entry of M_T is a number $[t]$, such that t is a node in T, t is a variable and $depth(t) = j$.
  — the (i, j)-th entry of M_T $(i \neq 0)$ is a number $[t]$, such that t is a node in T, t has arity $i + 1$ and $depth(t) = j$.

We deliberately specify $[.]$ only by its type in Definition 3. In ACL2(ml), this function is dynamically re-defined for every library and every given proof stage, as we are going to describe shortly in Definition 4. In practice, there will be a set of such functions computed in every session of ACL2(ml).

To make the feature extraction uniform across all ACL2 terms appearing in the library, the matrices are extended to cover terms up to arity n and tree-depth m. The parameters n and m can vary slightly; for all libraries considered in the paper $n = 5$ and $m = 7$ were sufficient, giving a feature vector size of 49 – a small size compared to sizes up to 10^6 in sparse approaches [18, 22, 23]. Having a clustering algorithm working for small sets of examples is crucial for the technique of recurrent clustering we implement for ACL2(ml).

Recurrent Clustering. As discussed above, the function [.] influences results of ACL2(ml) proof-pattern recognition; and the computation of [.] needs to be sensitive to the similarities that exist between the symbols appearing in the proofs. As ACL2 is a functional language, every entry in a term tree is necessarily itself defined in ACL2. This symbol definition can itself be clustered against other definitions used by the library; and the process can be repeated recursively to include all the necessary standard library definitions. This is how the feature extraction becomes a part of **recurrent clustering** in ACL2(ml).

We first define some essential clustering parameters. ACL2(ml) connects automatically to clustering [2] algorithms available in Weka (K-means, Farthest-First and E.M.). Clustering techniques divide data into n groups of similar objects (*clusters*), where the value of n is a learning parameter provided by the user. Increasing the value of n makes the algorithm separate objects into more classes, and, as a consequence, each cluster will contain fewer examples. There is a number of heuristics to determine the optimal value of n; ACL2(ml) has its own function to dynamically adjust n for every run of clustering using an auxiliary *granularity* parameter. Granularity can be varied by the user; and ranges between 1 and 5, low granularity produces big and general clusters while high granularity produces small and precise clusters (see the top row of Table 2). Given a granularity value g, the number of clusters n is given by the formula

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

The clustering algorithm assigns a *proximity value* to every term in a cluster. This ranges from 0 to 1, and indicates the certainty of the given example belonging to the cluster. ACL2(ml) shares with ML4PG some additional heuristics to ensure output quality, e.g., all experiments are run 200 times and only the ones with high frequencies are displayed to the user.

The choice of clustering algorithm, granularity and ACL2 libraries are accommodated in ACL2(ml) through a menu included in the Emacs interface. On user's demand, the ACL2(ml) interface displays families of related theorems; we call this process *Theorem Clustering*. In addition, ACL2(ml) clusters all the library definitions in the background every time a new definition is introduced. We call this process *Definition Clustering*.

The main reason for distinguishing theorem clustering and definition clustering is as follows. Theorem clustering is the ultimate goal of the proof mining here, but the feature tables for the theorems depend on the numeric representation of the symbols appearing in the theorems (see Definition 3 and Table 1). These symbols are normally defined within the libraries one uses, or else imported from CLISP. The following definition proceeds inductively on the type of symbols appearing in ACL2 definitions.

Definition 4 (Function [.]**).** *Given the nth term definition of the library (call the term t), a function* [.] *is inductively defined for every symbol s in t as follows:*
− [s] = i, if s is the ith distinct variable in t (note that all formulas are implicitly universally quantified in ACL2);
− [s] = −[m], if t is a recursive definition defining the function s with measure

* Type recognisers ($r = \{$symbolp, characterp, stringp, consp, acl2−numberp, integerp, rationalp, complex−rationalp$\}$): $[r_i] = 1 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$ (where r_i is the ith element of r).
* Constructors ($c = \{$cons, complex$\}$): $[c_i] = 2 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.
* Accessors ($a^1 = \{$car, cdr$\}$, $a^2 = \{$denominator, numerator$\}$, $a^3 = \{$realpart, imagpart$\}$): $[a_i^j] = 3 + \frac{1}{10 \times j} + \frac{i-1}{100}$.
* Operations on numbers ($o = \{$unary $-/$, unary $--$, binary $-+$, binary $-*\}$): $[o_i] = 4 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.
* Integers and rational numbers: $[0] = 4.3$, $[n] = 4.3 + \frac{|n|}{10}$ (with $n \neq 0$ and $|n| < 1$) and $[n] = 4.3 + \frac{1}{100*|n|}$ (with $n \neq 0$ and $|n| \geq 1$).
* Boolean operations ($b = \{$equal, if, $<\}$): $[b_i] = 5 + \sum_{j=1}^{i} \frac{1}{10 \times 2^{j-1}}$.

Fig. 4. Formulas to compute the value of function [.] for the ACL2 functions imported from CLISP. The above formulas serve to assign closer values to the functions within each of the six above groups, and more distant numbers across the groups – thus distinguishing the groups unambiguously.

function m automatically assigned by the ACL2;
$-$ $[s] = k$, if s is a function imported from CLISP; and $[s] = k$ in Figure 4;
$-$ $[s] = 5 + 2 \times j + p$, where C_j is a cluster obtained as a result of definition clustering with granularity 3 for library definitions 1 to $n-1$, $s \in C_j$ and p is the proximity value of s in C_j. (Note that a cluster in definition clustering is given by a set of terms; and the default granularity 3 generally provides a good balance between the size of clusters and their precision.)

Note the recurrent nature of clustering in Definition 4, with symbol numbering for the nth term depending on the clustering results for previous $n-1$ terms. As the above definition implies, the function [.] is adaptive, and is recomputed automatically when new definitions (and hence new symbols) are introduced. The motivation behind the various parameters of Definition 4 is as follows:
$-$ *Variables.* The variable encoding reflects the number and order of unique variables appearing in the term, note its correspondence to the De Bruijn indexes.
$-$ *Recursive case.* For every recursive function s, ACL2 assigns a termination measure function m. So, m necessarily exists for all recursive definitions and implicitly contains some "type" information (e.g. the measure for the function `rev` is the length of its input and the measure for `fact` is the value of its input). It has a negative value in order for feature values to distinguish the occurrence of the inductive symbol being currently defined from occurrence of any external functions invoked in the body of the term. The value $-[m]$ identifies all inductively defined symbols with the same ACL2 termination measure function.
$-$ Finally, the formula $5 + 2 \times j + p$ assigns $[s]$ a value within $[5 + 2 \times j, 5 + 2 \times j + 1]$, depending on their statistical proximity p for that cluster – p always lies within $[0, 1]$. Thus, elements of the same cluster have closer values comparing to the values assigned to elements of other clusters or to the imported CLISP functions.

We finish this section with some examples of ACL2(ml) clustering. All examples are run with several clustering functions and a choice of statistical parameters, allowing us to evaluate whether the feature extraction method we present here is robust across a range of algorithms, see Table 2.

	Algorithm:	$g = 1$ $(n = 16)$	$g = 2$ $(n = 18)$	$g = 3$ $(n = 21)$	$g = 4$ $(n = 25)$	$g = 5$ $(n = 30)$
fib-fib-tail experiments (one relevant cluster)	K-means	$9^{a,b,c}$	$4^{a,b,c}$	$3^{a,c}$	2^a	2^a
	E.M.	$16^{a,b,c}$	$16^{a,b,c}$	$9^{a,b,c}$	$4^{a,b,c}$	$4^{a,b,c}$
	FarthestFirst	$12^{a,b,c}$	$12^{a,b,c}$	$10^{a,b,c}$	$5^{a,b,c}$	$4^{a,b,c}$
	Algorithm:	$g = 1$ $(n = 11)$	$g = 2$ $(n = 12)$	$g = 3$ $(n = 14)$	$g = 4$ $(n = 16)$	$g = 5$ $(n = 20)$
revrev experiments (one relevant cluster)	K-means	$22^{d,e}$	$22^{d,e}$	$20^{d,e}$	$11^{d,e}$	$3^{d,e}$
	E.M.	$29^{d,e}$	$25^{d,e}$	$25^{d,e}$	$23^{d,e}$	$19^{d,e}$
	FarthestFirst	$45^{d,e}$	$34^{d,e}$	$30^{d,e}$	$26^{d,e}$	$15^{d,e}$

Table 2. *Results of Clustering experiments in ACL2(ml)* for a choice of algorithms and granularities. When granularity g is chosen by the user, ACL2(ml) dynamically calculates the number n of clusters; the table shows the size of one most relevant cluster that ACL2(ml) displays to the user in each case. **Top table**. Experiments for Theorem `fib-fib-tail` using JVM library (150 lemmas). We mark if the cluster contains (in addition to Theorem `fib-fib-tail`): a) Theorem `fact-fact-tail`, b) Theorem `power-power-tail`, c) other theorems related to the equivalence of recursive and tail-recursive functions. **Bottom table**. Experiments for Theorem `revrev` using Lists library (100 lemmas). All the clusters contain theorems d) `sortsort` and e) `int-x-x` (in addition to `revrev`). Note the stable performance of our feature extraction algorithm across several algorithms and parameters.

Example 4. Using the *JVM* library, ACL2(ml) detects lemmas similar to theorem `fib-fib-tail` (see Figure 2). Table 2 shows the sizes of clusters that ACL2(ml) will display for this theorem for various clustering algorithms and granularities. In Table 2, we can see a clear pattern: `fib-fib-tail` is consistently grouped with other theorems related to the equivalence of recursive and tail-recursive functions – this is done with all variations of learning algorithms and granularities, albeit with varied degree of precision. For this set of examples and this stage of the proof, the feature extraction function [.] returned values:
$[\texttt{fact}] = 12.974$, $[\texttt{power}] = 12.973$, $[\texttt{fib}] = 12.618$.
$[\texttt{helper-fact}] = 16.961$, $[\texttt{helper-power}] = 16.967$, $[\texttt{helper-fib}] = 16.431$.
$[\texttt{fact-tail}] = 18.970$, $[\texttt{power-tail}] = 18.969$, $[\texttt{fib-tail}] = 18.735$.
Note that the numbers above correspond to our intuitive grouping in Figure 2.

Table 2 also shows the lemmas of library *Lists* that are similar to `revrev`. In this case, the most precise cluster is detected by K-means with $g = 5$, but we can notice that `revrev` is always clustered with theorems `sortsort` and `int-x-x`.

4   Lemma Discovery in ACL2(ml)

One of the motivations of this research is to provide the ACL2 users with an efficient interactive lemma suggestion mechanism when automated proof search fails. The symbolic side of ACL2(ml) uses analogical reasoning to efficiently produce lemmas which are relevant to the current conjecture. In particular, it automatically attempts to suggest lemmas which are characterised as *eureka lemmas* – i.e. ones whose invention is mathematically creative and difficult to automate. In Section 3, we described how ACL2(ml) employs statistical machine-learning to determine similarity of theorem statements based on the term structure. In this section, we show how ACL2(ml) uses this information to construct analogous lemmas – this greatly reduces the search space for lemma discovery.
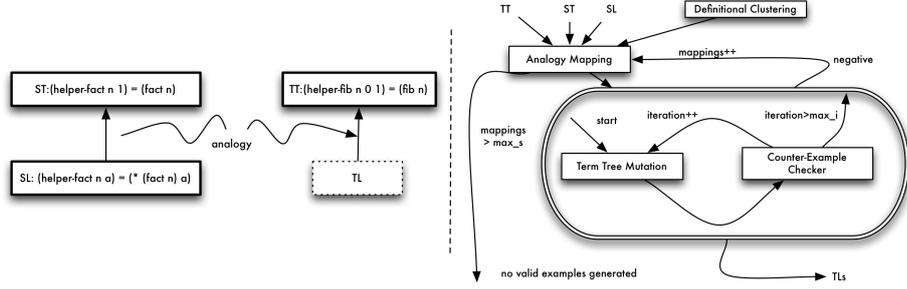
Fig. 5. *Left: The source theorem and lemma along with the target theorem for which we seek an analogous lemma. Right: overview of the process of generating analogical lemmas. Here the label* `iteration` *is the iteration step and* `mappings` *is a count of the number of analogy mappings.*

We introduce the following terminology: A *Target Theorem (TT)* is a theorem currently being attempted in ACL2, but requiring user's intervention. A *Source Theorem (ST)* is a theorem which has been suggested as similar to the TT by the statistical ACL2(ml). A *Source Lemma (SL)* is a user-supplied lemma required for proving the source theorem. The symbolic side of ACL2(ml) groups (potentially multiple) statistical suggestions into ST and SL pairs, each being evaluated in turn. The process then outputs some *Target Lemmas (TLs)* — these lemmas are analogical to some SL and not falsified by counter-example checking.

Example 5. Let us consider the case shown by Figure 5, where a user of ACL2 wants to prove the TT[4]: `(helper-fib n 0 1) = (fib n)`. As described in Section 3 and Table 2, ACL2(ml) suggests the closest analogy to be:
`(helper-fact n 1) = (fact n)`. Only one user-defined auxiliary lemma is used in the proof of this ST: `(helper-fact n a) = (* a (fact n))`. The job of the lemma analogy process is to construct the corresponding TL for the `fibonacci` example, this will be our running example throughout this section.

The overall process for lemma analogy is shown in Figure 5. It has two main components: *Analogy Mapping* and *Term Tree Mutation*. Analogy Mapping calculates which symbols could be analogical to each other using the definition clustering techniques from Section 3. Term Tree Mutation then uses this information to construct candidate target lemmas analogical to a given SL. Symbols belonging to the background theory, i.e. symbols shared between the source and target, are not changed by Analogy Mapping.

Definition 5 (Analogy Mapping $\mathcal{A}$). *For all symbols $s_1, \ldots, s_n$ occurring in the current ST, the set of admissible analogy mappings is the set of all mappings $\mathcal{A}$ such that*
- $\mathcal{A}(s_i) = s_i$ *for all shared background symbols; otherwise:*

---

[4] To ease readability, we use infix notation instead of ACL2 notation, e.g. `t1 = t2` instead of `(equal t1 t2)`.

- $\mathcal{A}(s_i) = s_j$ *for all combinations of* $i, j \in 1 \dots n$, *such that* s_i *and* s_j *belong to the same cluster in the last iteration of definition clustering.*

Example 6. For our running example, the shared background theory includes symbols {`+`, `*`,`-`, `1`, `0`}. We thus get a mapping:
$\mathcal{A} = \{$`fact` $\mapsto$ `fib`, `helper-fact` $\mapsto$ `helper-fib`, `+` $\mapsto$ `+`, `1` $\mapsto$ `1` $\dots \}$

Term Tree Mutation. Given an analogy mapping, candidate lemmas are constructed by mutating the term tree of the SL. IsaScheme [21] works similarly, by defining *schemes* which determine the shape of a generated term. Here, we extend this idea by iteratively allowing larger perturbations on this shape if no suitable candidate lemma has been found. We refer to this as *mutation*, an overview of the process is shown in Figure 5. The algorithm proceeds down three levels of increasing term tree mutation. After each iteration, we test the validity of the set of generated equations using a counter-example checker. If no candidate conjecture survives counter-example checking, mutation proceeds to the next iteration. The Term Tree Mutation algorithm for the case of equational lemmas is shown in Figure 6, however note that each sub-routine work on arbitrary terms and the mutation algorithm can thus trivially be generalised. The first iteration, *Tree Reconstruction*, replaces symbols in the SL with their analogical counterparts. The second iteration, *Node Expansion* further mutates the term, by synthesising small terms (max depth 2) in place of variables. Finally, the last iteration, *Term Tree Expansion* similarly adds new term structure, but on the top-level of the term.

Iteration 1 − Tree Reconstruction. The first iteration recursively replaces symbols in the source term by their analogical counterpart specified by $\mathcal{A}$. If the analogous function has fewer arguments than the source, we simply ignore excess arguments. If it has more arguments, we insert variables in the free positions.

Example 7. The running example has SL (`helper-fact n a`) = (`* a (fact n)`) with $\mathcal{A}$ as in Example 6. `TreeRec` performs the replacements to produce sets of candidate left- and right-hand sides, including for instance, the potential left-hand side (`helper-fib n n1 a`) and the right-hand sides (`* a (fib n)`) and (`* n1 (fib a)`) – as well as other variants with variables in different order. Note that we introduce a new variable `n1` as `helper-fib` has one more argument than its analogical counterpart `helper-fact`. In this case, no combination of left- and right-hand sides yield an equational theorem.

Iteration 2 − Node Expansion. If Tree Reconstruction fails to produce any conjecture passing counter-example testing, the second iteration synthesises terms which are allowed to replace variables, thus growing the term tree from the leaves. These synthesised terms are however limited to depth 2 and built from the shared function symbols, to keep the search space tractable.

Example 8. In our running example, we look at the new terms generated from the right-hand side of the SL. Suppose we start from one candidate from the previous iteration: (`* n1 (fib a)`). Node Expansion will consider all possible ways

```
TTMutation
Input:  An analogy mapping A, a set of shared symbols F, and an equational source
        term tl = tr.

 1. Compute Tree Reconstructions giving sets of candidate left- and right hand sides.
L1 = TreeRec(A, tl)
R1 = TreeRec(A, tr)

 Counter-example check all pairs of terms as candidate equations.
Res = TestAllEqs(L1, R1)
if not(Res = []) then return Res
 If no candidate lemma found, continue to next iteration.
else

 2. Compute Node Expansions
L2 = map (NodeExp F) L1
R2 = map (NodeExp F) L2

 Counter-example check all pairs of terms generated so far as candidate equations.
Res = TestAllEqs(L1++L2, R1++R2)
. . .

 3. Compute Tree Expansions
L3 = (TreeExp F) L2
R3 = (TreeExp F) R2

return TestAllEqs (L1++L2++L3, R1++R2++R3)
```

Fig. 6. Term Tree Mutation. The algorithm proceeds through three iterations of increasingly mutating the source term. This work is performed by the functions `TreeRec`, `NodeExp` and `TreeExp`. The left- and right hand sides of the original equation are mutated separately and after each step all combinations of equalities are counter-example tested (by the function `TestAllEqs`).

of replacing the variables `a` and/or `n1` by terms built from shared background theory functions {`+`, `*`, `-`}, applied to available variables, {`a`, `n`, `n1`}, and constants, {`1`, `0`}. The set of potential right-hand sides will now include terms such as `(* n1 (fib (- n 1)))` (having replaced `a` by `(- n 1)`), as well as many more similar alternatives.

Iteration 3 – Term Tree Expansion As opposed to Iteration 2, which introduced new term structure at the leaves, this phase allows for insertion of term structure at the top level. Term Tree Expansion considers all terms synthesised so far and explores how they can be used as arguments to shared background theory functions.

Example 9. For the running example, we consider the whole set of candidate terms for the right-hand side. We consider all terms that can be built giving some combination of these as arguments to the background theory functions {`+`, `*`, `-`}. For instance, one of the new terms that is built in this iteration is `(+ (* n (fib (- n1 1))) (* a (fib n1))`, having added a top-level `+`. The counter-example checker does not find a counter-example for the conjecture `(helper-fib n n1 a) = (+ (* n (fib (- n1 1))) (* a (fib n1))`. Hence, the result is shown to the user as a suggestion of an auxiliary lemma to employ in the current proof.

5   Evaluation and Conclusions

We have presented ACL2(ml), an ACL2 extension combining statistical machine learning, to detect proof patterns, with symbolic techniques for generating new auxiliary lemmas. All software and extensive evaluation results are available from [10]. Both sides of ACL2(ml) are original techniques on their own. However, comparing ACL2(ml) with other alternative tools, two of its features distinguish it from all other approaches: its methods of generating the proofhints *interactively* and in real-time, in response to the user's call; and secondly, its flexible environment for integration of statistical and symbolic techniques. We finish with an evaluation of the system.

Scalability. The ACL2 proofs libraries can grow very big, especially in industrial scenarios. The statistical ACL2(ml) tool works well with libraries of varied sizes and complexities; and does not need any fine-tuning when the user adds more libraries. To illustrate this, we increased by 6.64 times the data set used in Example 4 and Table 2 (by adding as noise lemmas coming from standard ACL2 libraries; and also the ones presented in [19] – the data set contains 996 theorems). Table 3 shows that ACL2(ml) still finds precise clusters; e.g., it still finds a cluster containing exactly the three theorems proving equivalence of recursive and tail-recursive functions similar to `fib-fib-tail`. Similar results are shown with respect to other functions from e.g. JVM library, see [10].

	$g = 1$ $(n = 110)$	$g = 2$ $(n = 124)$	$g = 3$ $(n = 142)$	$g = 4$ $(n = 166)$	$g = 5$ $(n = 199)$
fib-fib-tail	$57^{a,b,c}$	$50^{a,b,c}$	$25^{a,b,c}$	$8^{a,b,c}$	$4^{a,b,c}$

Table 3. *Results of Clustering experiments in ACL2(ml)* for $k-$means clustering algorithm and a choice of granularities. Experiments for Theorem `fib-fib-tail`. We mark if the cluster contains (in addition to Theorem `fib-fib-tail`): *a*) Theorem `fact-fact-tail`, *b*) Theorem `power-power-tail`, *c*) theorems related to the equivalence of recursive and tail-recursive functions. The size of the data set is 996 theorems coming from the JVM library [12], the standard ACL2 libraries: Lists, powerlists and sorting; and a library related to the formalisation of a computer algebra system presented in [19].

Usability of statistical suggestions. The statistical side of ACL2(ml) is not deterministic; but it is possible to empirically evaluate the usability of the statistical results produced by ACL2(ml). Using the data set containing 996 theorems presented in Table 3, and considering the clusters obtained with granularity 5, we notice that: 37% of clusters identified by ACL2(ml) can be directly used by the Lemma Analogy tool of ACL2(ml) to mutate lemmas. Additionally, 15% of clusters contain theorems that use the same lemmas in their proofs, another 15% of clusters consist of theorems that are used in the proofs of other theorems of the same cluster, and about 19% of clusters contain basic theorems whose proofs are similar, but based on simplification, and hence unsuitable for Lemma Analogy application. Only 14% of clusters do not show a clear correlation that could be reused.

Modularity. ACL2(ml) provides a flexible environment for integrating statistical and symbolic machine-learning methods. The statistical parameters can be easily tuned to provide a better input for the Analogy tool.

Example 10. Consider the generation of auxiliary lemmas for `fib-fib-tail` and `revrev`. The auxiliary lemma for `fib-fib-tail` is generated from the lemma marked with either *a* or *b* in Table 2. Other lemmas in this cluster can be considered as noise. In this case, the granularity and algorithm choice are not particularly relevant because the auxiliary lemma can be created from all the clusters and the search space is small. However, the correct auxiliary lemma is generated faster when increasing the granularity parameter.

In the case of `revrev`, the relevant clusters are bigger than in the case of `fib-fib-tail` and the relevant lemma for `revrev` is only generated from theorem `sortsort`. Then, the noise can be reduced by setting ACL2(ml) to the highest granularity 5 with K-means algorithm, which reduces the cluster to 3 lemmas.

All our running examples were based on the cases when the proofs of the theorems required introduction of only one user-defined auxiliary lemma; and hence the analogy tool concentrated only on them. However, in the general case, the similar theorems found by statistical ACL2(ml) can rely on several auxiliary lemmas. To help the Analogy tool decide which of them should be used for new lemma generation, the statistical tool can be applied recursively to cluster these auxiliary lemmas; and thus find those more likely to result in a good analogy.

Lemma Discovery. The Analogy algorithm is designed to be fast and reduce the combinatorial explosion introduced by typical theory exploration techniques. By starting from an exemplar source lemma, we use mutation to perturb the term. We compare our tool with QuickSpec [5] – a state of the art system for generating equational conjectures about Haskell programs. While QuickSpec is designed to suggest a number of conjectures about a given set of symbols, the lemma analogy system aims at just discovering a few lemmas relevant to some particular proof attempt.

The ACL2(ml) approach is more suitable for integration into an ITP, as analogy can reduce the search space, making lemma discovery tractable in theories where QuickSpec runs out of memory. QuickSpec works by generating all terms up to a given depth (default is 3) and then use testing to evaluate the terms and divide them into equivalence classes. These represent a large number of equations so the set is pruned before displaying it to the user in an attempt to only show a small set of interesting equations. Table 4 shows the results of all combinations of source and target theorems from the examples used on the natural numbers examples introduced into ACL2(ml). QuickSpec takes as input only the function symbols from the target theorems for each run. Both systems produce results instantly, except in the cases where QuickSpec runs into exponential blowups. The Analogy tool usually returns one lemma relevant to the TT, whereas QuickSpec may return a very large number of results (some of which are not valid), which is not ideal for displaying to a user of an interactive system. It also runs out of memory on examples involving functions with many arguments and a higher degree of commutativity, such as the tail recursive version of multiplication (as the number of terms within its depth limit increases). These kind of theories are generally problematic for any theory exploration system relying merely on term

		Target							QuickSpec		
		$fact$	$power$	$expt$	sum	sum_sq	$mult$	fib	Lemma	Valid	Invalid
Source	$fact$	-	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_2$	$\checkmark_1$	×	10	5
	$power$	$\checkmark_1$	-	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_2$	$\checkmark_1$	×	17	4
	$expt$	$\checkmark_1$	$\checkmark_1$	-	$\checkmark_1$	$\checkmark_1$	$\checkmark_2$	$\checkmark_1$	×	OoM	OoM
	sum	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	-	$\checkmark_1$	$\checkmark_2$	$\checkmark_1$	×	7	2
	sum_sq	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	-	$\checkmark_2$	$\checkmark_1$	×	7	1
	$mult$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	$\checkmark_1$	-	$\checkmark_1$	×	200	20
	fib	$(\checkmark_2)$	$(\checkmark_2)$	×	×	×	×	-	×	OoM	OoM

Table 4. *Results of lemma analogy in comparison with QuickSpec. $\checkmark_n$ denotes a success with n returned lemmas, and $(\checkmark_n)$ denotes a valid generated lemma which is not applicable to the TT. × denote failure to find exactly the desired lemma, while OoM denotes that the system ran out of memory. The numbers correspond to the number of formulae generated by QuickSpec*

generation, e.g. [4, 13, 21]. Despite their heuristics, the search space grows too big. The integration in an ITP allows us to use information about the current proof attempt, and thus navigate these theories.

Like proof-critics [1], which is a technique for lemma discovery in automated provers, we use information from the current proof-attempt to guide the search for auxiliary lemmas. However, while proof-critics only analyses the current proof attempt, ACL2(ml) also learns from all previous proofs. Some proof-critics are also closely reliant on particular proof-techniques, such as rippling, while ACL2(ml) is independent of this.

Limitations and Further Work. Below, we summarise the limitations of the system and indicate directions of further work. Although the statistical side of ACL2(ml) can process and cluster any proof and lemma shape, the symbolic analogy tool cannot handle all possible analogical cases (and no algorithmic approach could do that in principle).

Different patterns. Statistical ACL2(ml) groups in the same clusters theorems `revrev`, and `int-x-x`; however, the lemmas used in `revrev` cannot be mutated to generate any of the lemmas needed in the proof of `int-x-x`.

Smaller lemmas. The lemma analogy tool currently only adds term structure; therefore, it cannot generate smaller lemmas. E.g. Lemmas `fact-fact-tail` and `fib-fib-tail` are in the same cluster (see Table 2); the analogy tool succeeds in generating the lemma `fib-fib-tail` given the source lemma `fact-fact-tail`, but fails to go the other way around. This would require removing term structure, i.e. the inverse operation of Term Tree Expansion (see Section 4).

Conditional lemmas. Apart from type recogniser conditions (e.g. `natp` is the type recogniser for natural numbers), several ACL2 lemmas need additional conditions, which, at the moment, are not generated by the symbolic ACL2(ml). Discovering appropriate conditions for generated lemmas is a difficult problem for theory exploration systems.

New definitions. Another big challenge in lemma discovery is invention of new concepts. The proof of Lemma `int-x-x` needs a new concept called `subp`

which is the recogniser for subsets of lists – `(subp x y)` returns `true` or `false` according to whether every element of `x` is an element of `y`.

References

1. D. Basin, A. Bundy, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.
2. C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.
3. B. Buchberger et al. Theorema: towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):2006, 470–504.
4. K. Claessen et al. Automating inductive proofs using theory exploration. In *CADE-24*, volume 7898 of *LNCS*, pages 392–406, 2013.
5. K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *TAP'10*, volume 6143 of *LNCS*, pages 6–21, 2010.
6. S. Colton. The HR Program for Theorem Generation. In *CADE-18*, volume 2392 of *LNCS*, pages 285–289, 2002.
7. J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from previous proof experience: A survey. Technical report, Technische Universitat Munchen, 1999.
8. J. Denzinger and S. Schulz. Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts. *Inform. and Comput.*, 162(1-2):59–79, 2000.
9. H. Duncan. *The use of Data-Mining for the Automatic Formation of Tactics.* PhD thesis, University of Edinburgh, 2002.
10. J. Heras et al. ACL2(ml): downloadable programs, manual, examples, 2013. `http://staff.computing.dundee.ac.uk/jheras/acl2ml`.
11. S. Hetzl, A. Leitsch, and D. Weller. Towards algorithm cut-introduction. In *LPAR-18*, volume 7180 of *LNCS*, pages 228–242, 2012.
12. J Moore. Proving theorems about Java and the JVM with ACL2. In *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, 2003.
13. M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *JAR*, 47(3):251–289, 2011.
14. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
15. M. Kaufmann and J S. Moore. How to Prove Theorems Formally, 2005. `http://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/main.ps`.
16. M. Kaufmann and J S. Moore. ACL2 version 6.2, 2013. `http://www.cs.utexas.edu/users/moore/acl2/`.
17. E. Komendantskaya, J. Heras, and G. Grov. Machine Learning for Proof General: interfacing interfaces. *ENTCS*, 118:15–41, 2013.
18. D. Kühlwein et al. Overview and evaluation of premise selection techniques for large theory mathematics. In *IJCAR'12*, volume 7364 of *LNCS*, pages 378–392, 2012.
19. F. J. Martín-Mateos et al. ACL2 verification of simplicial degeneracy programs in the Kenzo system. In *CICM'09*, volume 5625 of *LNCS*, pages 106–121, 2009.
20. R. L. McCasland, A. Bundy, and P F. Smith. Ascertaining mathematical theorems. *ENTCS*, 151(1):21–38, 2006.
21. O. Montano-Rivas et al. Scheme-based theorem discovery and concept invention. *Expert Syst. Appl.*, 39(2):1637–1646, 2012.
22. E. Tsivtsivadze, J. Urban, H. Geuvers, and T. Heskes. Semantic graph kernels for automated reasoning. In *SDM'11*, pages 795–803, 2011.
23. J. Urban et al. Malarea sg1- machine learner for automated reasoning with semantic guidance. In *IJCAR'08*, volume 5195 of *LNCS*, pages 441–456, 2008.