ACL2(ml)

Heras, Jónathan; Komendantskaya, Ekaterina

Published in:
Electronic Proceedings in Theoretical Computer Science

DOI:
10.4204/EPTCS.152.5

Publication date:
2014

Licence:
CC BY

Document Version
Publisher's PDF, also known as Version of record

ACL2(ml): Machine-Learning for ACL2*

Jónathan Heras
School of Computing, University of Dundee, UK
jonathanheras@computing.dundee.ac.uk

Ekaterina Komendantskaya
School of Computing, University of Dundee, UK
katya@computing.dundee.ac.uk

ACL2(ml) is an extension for the Emacs interface of ACL2. This tool uses machine-learning to help the ACL2 user during the proof-development. Namely, ACL2(ml) gives hints to the user in the form of families of similar theorems, and generates auxiliary lemmas automatically. In this paper, we present the two most recent extensions for ACL2(ml). First, ACL2(ml) can suggest now families of similar function definitions, in addition to the families of similar theorems. Second, the lemma generation tool implemented in ACL2(ml) has been improved with a method to generate preconditions using the guard mechanism of ACL2. The user of ACL2(ml) can also invoke directly the latter extension to obtain preconditions for his own conjectures.

1 Introduction

ACL2 is a theorem prover that has been successfully applied in the formalisation of industrial problems [3,18], and also used for pedagogical purposes [11,23]. Even if there is a considerable difference in the difficulty of the problems that are tackled in these two contexts, there are some common challenges that are faced by experts and novice users of ACL2 alike: re-usability of libraries, discovery of auxiliary lemmas, and comparison of similarities between theorems and definitions. In the industrial scenario, the coordination of the members of a team is also a challenge: each user has his own definitions, theorems and proof-style, making the collaborative proof-development difficult.

These challenges are partially addressed by the searching tools already available in ACL2. ACL2 provides several alternatives to browse its documentation, and search definitions and theorems in the ACL2 books. The ACL2+books combined manual can be accessed online [8], using the Emacs-based ACL2-Doc browser, or using the ACL2 `:DOC` command at the terminal. In addition, ACL2 books can be browsed from the community-books website [7].

The above tools can be used to search information about an ACL2 topic, access the documentation of a concrete theorem or definition, or search a keyword in the ACL2 books. Hence, these tools are useful if the user knows what he is searching for. When the user fails to define searching parameters exactly, it may be helpful to use statistical machine-learning to detect and identify relevant patterns. For example, consider the following scenarios:

1. a user in the middle of a proof does not know how to proceed, and wishes to see some similar development to extrapolate it to his particular case;
2. a user has the intuition that he is proving/defining something similar (or exactly the same) to one of his previous developments, but does not remember exactly where and how he did it; and,
3. a user in a team-development wants to know if any other member of the team has solved his current (or a similar) problem.

*This work was supported by the SICSA Industrial Proof of Concept grant "Machine-Learning for Industrial Theorem Proving", and EPSRC grants EP/J014222/1 and EP/K031864/1.

F. Verbeek and J. Schmaltz (Eds.): ACL2 Workshop 2014 (ACL2'14).
EPTCS 152, 2014, pp. 61–75, doi:10.4204/EPTCS.152.5

In these three situations, the current searching mechanisms of ACL2 are not useful because the user does not have exact parameters for search; however, it would be extremely helpful to have a tool that could automatically detect patterns across ACL2 books; capturing higher-level intuitions the user may have.

ACL2(ml) [14, 15] – a machine-learning extension for the Emacs interface of ACL2 – was created with that particular aim. ACL2(ml) uses statistical clustering to detect families of ACL2 theorems following the same pattern; and symbolic methods to automatically generate new lemmas. In particular, ACL2(ml) features the following main functions (see also Figure 1):

- The user works within the Emacs environment of ACL2, and has an option to call ACL2(ml) whenever he needs to see similar patterns.
- Based on the user's choice, ACL2(ml) compiles the chosen libraries, and extracts significant features from the terms of ACL2 theorems and definitions.
- ACL2(ml) 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 ACL2 user of the laborious step of post-processing the statistical results.
- If the user chooses to see only patterns related to his current theorem, ACL2(ml) would further filter the results and show the families of related theorems to the user.
- Additionally, and based on the families of similar theorems, ACL2(ml) generates lemmas that may help in the proof of a given theorem.
- The related theorems and generated auxiliary lemmas are displayed in a separate window.

An overview of the initial ACL2(ml) features is given in Section 2, the details of the different techniques implemented in ACL2(ml) are given in [15]. In this paper, we present the two most recent extensions of ACL2(ml): *definition clustering* and a *generator of preconditions*.

The initial ACL2(ml) was focused on finding families of similar theorems; however, discovering families of similar definitions can speed-up the proof-development and avoid redundancies in the code. For instance, ACL2(ml) can discover that a newly defined function was previously defined; in that case, the user can use the existing library definition and all its background theory instead of defining it from scratch. We present the definition clustering method and its applicability to find redundancies in the ACL2 books in Section 3.

The second extension that we introduce in this paper improves the lemma generation tool implemented in ACL2(ml). The method presented in [15] generates a set of conjectures that could be useful to prove a given theorem. These conjectures are automatically tested with a counterexample generator, and ACL2(ml) only suggests to the user the conjectures that are not falsified. Some of the conjectures are rejected because the necessary preconditions are missed. In Section 4, we explain the use of ACL2 guards to find preconditions for conjectures. This technique is incorporated into the lemma generation component of ACL2(ml); additionally, the user can use this extension to find preconditions of his own conjectures.

Finally, in Section 5, we conclude the paper. ACL2(ml) and the libraries used in this paper can be downloaded from [14].

2 Overview of ACL2(ml)

ACL2(ml) combines *statistical* and *symbolic* machine-learning techniques to help the ACL2 user during the proof development (cf. Figure 1). In this section, we present the main functionality that the initial ACL2(ml) offers to the user.

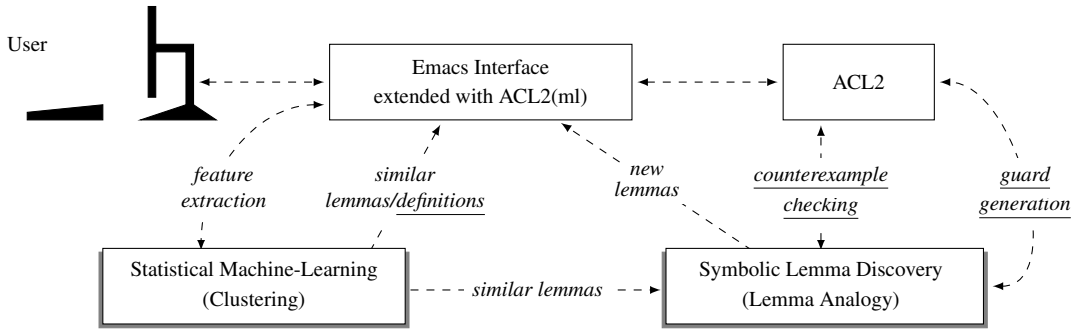


Figure 1: *Architecture of ACL2(ml).* The Emacs interface for ACL2 extracts important features from ACL2 theorems and definitions, connects to machine-learning software, clusters the theorems there, and sends the result (families of similar theorems) to the screen. In addition, for an unproved theorem T , it sends the cluster C of T to the lemma analogy tool, which in its turn generates auxiliary lemmas by analogy with auxiliary lemmas of the theorems in C. The underlined components of the diagram are the extensions presented in this paper.

Statistical machine-learning in ACL2(ml). ACL2(ml) uses (unsupervised) clustering algorithms [2] to find patterns in ACL2 theorems. Clustering algorithms divide data into n groups of similar objects (called clusters), where the value of n is a parameter provided by the user. In ACL2(ml), the value of n is automatically computed depending on the number of objects to cluster, and using the formula provided in [15]. Clustering methods find significant patterns among selected features. Hence, ACL2(ml) must have its own method to extract significant features from ACL2 terms.

We adopt the following standard terminology. We assume there is a training data set, containing some samples (or objects). *Features* are given by a set of statistical parameters chosen to represent all objects in the given data set. If n features are chosen, one says that object classification is conducted in an n-dimensional space. For this reason, most pattern-recognition tools will require that the number of selected features is limited and fixed. *Feature values* are rational numbers used to instantiate the features for every given object. If an object is characterised by n feature values, these n values together form a *feature vector* for this object. A function that assigns, to every object of the data set, a feature vector is called a *feature extraction function*.

Feature extraction [2] is a research area developing methods for discovery of statistically significant features in data. ACL2(ml) implements its own feature extraction method to collect statistical features from ACL2 terms. It captures the structure and dependencies of ACL2 terms by processing the whole ACL2 development. ACL2 term trees are central to this process.

Definition 2.1 (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 t1 ... tn)` *is represented by the tree with the root node labelled by* `f`*, and its immediate subtrees given by trees representing* `t1`, ..., `tn`.

The features collected from the term trees are given by a finite number of properties common to all possible trees: the term tree depth and the term arity of the function of a node. These parameters must be fixed at an arbitrary size. In the current standard version of ACL2(ml), we limit both parameters to 7. Thus, given a term t, we construct a 7×7 table, $[t]_M$, where the rows represent term-tree depth and the columns represent term arity (where the first column is reserved for variables) – i.e. the element $(0, j)$ of the table contains the variables of the term tree of depth j, and the element (i, j) contains the terms of depth j and arity $i-1$ (cf. Figure 3). Fixing the maximum depth and maximum arity of a node makes the

```
;; Multiplication
1 (defun mult (n m) (if (zp m) 0 (+ n (mult n (- m 1)))))
2 (defun helper-mult (n m a) (if (zp m) a (helper-mult n (- m 1) (+ n a))))
3 (defun mult-tail (n m) (helper-mult n m 0))
4 (defthm mult-mult-tail (implies (and (natp n) (natp m)) (equal (mult-tail n m) (mult n m))))
5 (defthm mult-helper-mult (implies (and (natp n) (natp m) (natp a))  (equal (helper-mult n m a) (+ a (mult n m)))))

;; Exponentiation
1 (defun expt (n m) (if (zp m) 1 (* n (expt n (- m 1)))))
2 (defun helper-expt (n m a) (if (zp m) a (helper-expt n (- m 1) (* n a))))
3 (defun expt-tail (n m) (helper-expt n m 1))
4 (defthm expt-expt-tail (implies (and (natp n) (natp m)) (equal (expt-tail n m) (expt n m))))
5 ???

;; 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))))
5 (defthm fact-helper-fact (implies (and (natp n) (natp a))  (equal (helper-fact n a) (* a (fact 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))))
5 ???
```

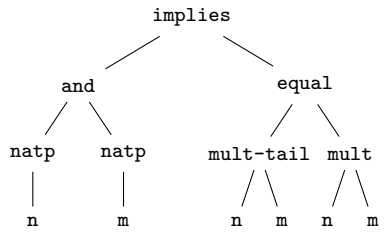
Figure 2: *ACL2 definitions and theorems.* 1: recursive arithmetic functions. 2: helpers of tail-recursive arithmetic functions. 3: tail-recursive arithmetic functions. 4: equivalence theorems of recursive and tail-recursive functions. 5: auxiliary lemmas to prove theorems of Point 4.

feature extraction mechanism uniform across all ACL2 terms appearing in the libraries – we may lose some information if pruning is needed, but the chosen size works well for most terms appearing in ACL2 libraries, and increasing the size will introduce sparsity in the feature matrices making the clustering process more difficult.

Example 2.2 *Consider the theorem* `mult-mult-tail` *given in Figure 2, its term tree is depicted in Figure 3. This term depends on other terms:* `implies`, `natp`, `equal` *and so on. To extract the feature table for* `mult-mult-tail`*, see Figure 3, we need to know the feature values of those functions comprised in this theorem, and this is achieved using the function* $[.]$.

The injective function $[.]$: ACL2 functions $\to \mathbb{Q}$ is dynamically recomputed as an ACL2 development progresses. This function is sensitive to the structure and dependencies of functions, and it assigns close values to similar functions using a *recurrent clustering* process [15]. Namely, given a function f, ACL2(ml) constructs the feature table $[f]_M$ from the definition of f, clusters it against the rest of the definitions of the library, and finally assigns a unique rational number, $[f]$, to the function depending on its cluster (the values of $[.]$ for the functions in the same cluster will be close, and functions in different clusters will have more distant values, see [15] for the concrete formula). In order to construct the feature table $[f]_M$, we need to know the feature values of the functions comprised in f, and this in turn can be done by clustering their definitions, and extracting their feature values — in the case of recursive (or mutually recursive) functions, the value for the recursive call is fixed and different to the values assigned from the clustering process. This recurrent clustering process continues up to the ACL2's built-in functions whose numerical values were pre-defined in advance. Thanks to this *recurrent clustering* process, the function $[.]$ assigns close values to similar functions (based on the structure and function calls of their definitions), and more distant values to unrelated functions.

Example 2.3 *For the functions presented in Figure 2, the function* $[.]$ *returned values:*
$[\texttt{fact}] = 12.974$, $[\texttt{fib}] = 12.618$, $[\texttt{mult}] = 10.965$, $[\texttt{expt}] = 10.959$,
$[\texttt{helper-fact}] = 16.961$, $[\texttt{helper-fib}] = 16.431$, $[\texttt{helper-mult}] = 16.548$, $[\texttt{helper-expt}] = 16.507$,



	variables	arity 0	arity 1	arity 2
td0	0	0	0	[implies]
td1	0	0	0	[and,equal]
td2	0	0	[natp,natp]	[mult-tail,mult]
td3	[n,m,n,m,n,m]	0	0	0

Figure 3: *Term tree and a fragment of the feature table of theorem* `mult-mult-tail` *from Figure 2.*

$[\texttt{fact-tail}] = 18.970$, $[\texttt{fib-tail}] = 18.735$, $[\texttt{mult-tail}] = 18.699$, $[\texttt{expt-tail}] = 18.640$.

The construction of feature tables is a process that runs in the background of ACL2(ml). When the user asks ACL2(ml) to show families of similar theorem statements, ACL2(ml) initialises the recurrent clustering procedure by using a set of Emacs-Lisp scripts that communicate with clustering algorithms implemented in the Weka machine-learning interface [13]. When the recurrent clustering cycle is completed, ACL2(ml) uses another set of scripts to cluster the statement of theorems, and post-processes the final clustering statistics showing the user only the best "guess" of similar theorems.

Example 2.4 *Let us consider a library that contains theorems about the equivalence of several recursive and tail-recursive arithmetic functions including the results presented in Figure 2. In this library, we have some theorems that have been proven (theorems about multiplication and factorial functions in Figure 2) and other theorems are still unproven (exponentiation and Fibonacci theorems in Figure 2). Using this library, ACL2(ml) will show a message with clusters of similar theorems; in particular, it detects that the theorems* `expt-expt-tail` *and* `mult-mult-tail` *belong to the same cluster, and theorems* `fib-fib-tail` *and* `fact-fact-tail` *form another cluster. We can use this information to extrapolate the auxiliary lemmas that were used in the proof of multiplication and factorial to complete the proofs of exponentiation and Fibonacci respectively.*

The user may prefer the statistical hint to be related to the current theorem that he is trying to prove, as in Example 2.4, or give information about patterns arising in the theorems of a library irrespective of the current theorem. The user may choose to data-mine only the current library, or a number of ACL2 books coming from different domains or different users. In addition, the user can configure the precision of clusters. All these options are accommodated within ACL2(ml), see [14] for a detailed description of the user interface.

Next, we can use the statistically discovered clusters of theorems to generate proof hints in the form of auxiliary lemmas.

Lemma Analogy in ACL2(ml). Symbolic tools for automatic discovery of lemmas, like IsaCoSy [17] and IsaScheme [22], synthesise new terms using different kinds of heuristics. These tools generate candidate conjectures which are then filtered through a counterexample checker. These methods have limits: they can be slow on large inputs due to the increase in the search space, and rely on having access to good counterexample finders for filtering of candidate conjectures.

The approach followed in ACL2(ml) to synthesise new terms relies on the statistically discovered clusters, and uses term trees clustered together to synthesise new theorems. Namely, in order to reduce

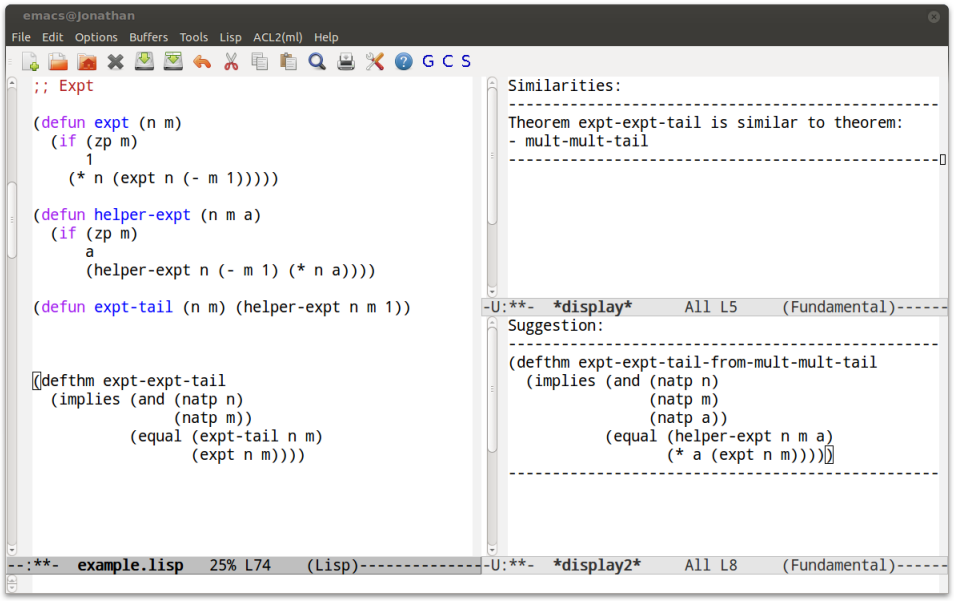


Figure 4: *Suggestions for the lemma expt-expt-tail.* The Emacs window has been split into three buffers: the left buffer keeps the current ACL2 development, the right-top buffer shows the lemmas that are similar to expt-expt-tail in the current development, and the right-bottom buffer shows the suggestion generated to prove lemma expt-expt-tail.

the search space, ACL2(ml) feeds the output of the clustering algorithms to a *mutation* tool. We adopt the following terminology. A *target theorem* (TT) is a theorem currently being attempted in ACL2, but requiring user's intervention in the form of auxiliary lemmas. 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 lemma required for proving the ST (the SLs are obtained inspecting the event summary printed by ACL2 after proving the ST). 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 counterexample checking.

Example 2.5 *In Example 2.4, for the case of exponentiation, we have:* `expt-expt-tail` *is the TT,* `mult-mult-tail` *is the ST, and* `mult-helper-mult` *is the SL. ACL2(ml) uses this information to generate the following TL by mutating the SL:*

```
(implies (and (natp n) (natp m) (natp a))
         (equal (helper-expt n m a) (* a (expt n m))))
```

This is the necessary lemma to prove the TT — Figure 4 shows how this feedback is presented in ACL2(ml).

The lemma analogy tool implemented in ACL2(ml) consists of three levels of increasing term tree mutation. After each iteration, ACL2(ml) tests the validity of the set of generated equations using a counterexample checker. If no candidate conjecture survives counterexample checking, mutation proceeds to the next iteration. The first iteration, *tree reconstruction*, replaces symbols in the SL with their analogical counterparts obtained from the TT. The second iteration, *node expansion*, further mutates the term, by synthesising small terms in place of variables. Finally, the last iteration, *term tree expansion*, similarly

adds new term structure, but on the top-level of the term. The algorithms for these three iterations were presented in [15].

In [15], the lemma analogy tool was implemented as a standalone OCaml application. This tool was automatically invoked from ACL2(ml) when the user requested the generation of auxiliary lemmas. In the current version of ACL2(ml), the lemma analogy tool has been reimplemented in Emacs Lisp making the tool more efficient. This approach avoids the translation of terms to a different language and does not require the communication with an external tool. ACL2(ml) checks the generated conjectures using the counterexample generator implemented for ACL2 Sedan [4].

We refer to the ACL2(ml) user manual [14] for a detailed description of how to use the tool.

3 Clustering Functions of the ACL2 Community Books

In this section, we present the first original contribution of this paper – extrapolation of the theorem clustering described in [15] to all ACL2 objects, including function definitions. In particular, we are interested in finding families of similar definitions; since the discovery of these families can avoid redundancies in the code and speed-up the proof-development.

Technically, we apply the same method of recurrent clustering, as described in the previous section, to all ACL2 definitions – the difference is that ACL2(ml) clusters function definitions, instead of clustering theorem statements, as final step after the recurrent clustering cycle. We will use this section to show how this extension of ACL2(ml) can help to analyse redundancies in the ACL2 books repository.

ACL2 books accumulate models, definitions, proofs and proof libraries that can be applied in new proof developments. However, due to the size of the ACL2 books repository, it is a challenge to trace them and find reusable concepts and proof ideas. For example, there is a total of 23290 definitions in the ACL2 distribution; then, it is unfeasible to manually detect patterns or redundancies arising in the function definitions. ACL2(ml) can perform this task automatically. Note that ACL2(ml) analyses definitions on the basis of their structure and their dependencies on other ACL2 functions; rather than on the concrete notation used in the definitions. Therefore, ACL2(ml) can detect cases when the same or similar notions are redefined with different names and using different notation.

Using its default configuration, ACL2(ml) discovers 981 clusters across the definitions of the ACL2 books repository in approximately 5 minutes. From the 981 clusters, we can distinguish the following classification of clusters (see also [14] for a supporting extended note about this experiment).

C.1. 39% of the clusters are related to a function that has been defined several times in different books of the same library.

Example 3.1 *The* `mergesort-exec` *function is implemented in the* `sets` *and* `sort` *books of the* `coi` *library. It is worth mentioning that in the definition of the* `sort` *book, ACL2 needs two explicit hints to accept the definition of* `mergesort-exec`*; on the contrary, in the* `sets` *book, these two hints are not necessary.*

C.2. 18% of the clusters are related to a function that has been re-defined in different versions of a library.

Example 3.2 *The* `rtl` *library includes* 5 *different versions, and several functions are repeated in all of them. For example, ACL2(ml) detects that the function* `logxor1` *is defined in* 4 *versions of this library.*

C.3. 22% of the clusters are related to a function that has been defined not only in different versions of a library, but also in different books of the same version of that library.

Example 3.3 *Most of the clusters that belong to this case come from the* `rtl` *library. As an example, ACL2(ml) finds the cluster containing the multiple definitions of* `fl`*, a function that has been defined* 108 *times across the different versions and books of the* `rtl` *library.*

C.4. 10% of the clusters contain functions that have been defined in the same library and that are closely related.

Example 3.4 *An example of this kind of cluster is found in the* `centaur` *library. ACL2(ml) discovers that the following two functions are in the same cluster.*

```
(defun nonsubset-witness (x y)
  (declare (xargs :guard (and (setp x) (setp y))))
  (if (empty x)
      nil
    (if (in (head x) y)
        (nonsubset-witness (tail x) y)
      (head x))))

(defun intersectp-witness (x y)
  (declare (xargs :guard (and (setp x) (setp y))))
  (if (empty x)
      nil
    (if (in (head x) y)
        (head x)
      (intersectp-witness (tail x) y))))
```

The similarity between these two theorems can be easily spotted (both functions share the same structure and background functions), but note that this similarity is automatically discovered by ACL2(ml).

C.5. 5% of the clusters are related to a function that has been defined several times in different libraries. These clusters are difficult to track manually since the same function is usually defined with different names; however, ACL2(ml) finds these clusters based on the structure and background information of the function definitions.

Example 3.5 *ACL2(ml) discovers that the libraries* `unicode` *and* `security` *define the same function to convert an ASCII character list into a Unicode string, but this function is called* `charlist=>ustring` *in the* `unicode` *library, and* `charlist-to-bytes` *in the* `security` *library.*

C.6. 6% of the clusters consist of functions that are similar and are defined across several ACL2 libraries.

Example 3.6 *The library* `coi` *contains, in two different books, the following two close definitions that recognise respectively if all the elements of a list are not integers, and if a list contains an integer.*

```
(defun all-not-integerp (x)
  (if (consp x)
      (if (integerp (car x))
          nil
        (all-not-integerp (cdr x)))
    t))

(defun some-integerp (x)
  (if (consp x)
      (if (integerp (car x))
          t
        (some-integerp (cdr x)))
    nil))
```

In the above classification, ACL2(ml) discovers two kinds of clusters across ACL2 books: clusters related to a function that has been redefined in several books, and clusters of similar functions. The clusters that are classified as **C.1**–**C.3** could be found by searching function definitions with the same name. However, it is worth remarking that ACL2(ml)'s method goes beyond keyword searching and discovers functions that are defined with different names (clusters classified as **C.5**) and functions that are close related (clusters classified as **C.4** and **C.6**). In particular, the main properties that distinguish ACL2(ml) pattern search from the searching tools in ACL2 are:

- the user does not have to know or provide any searching parameter;
- the discovered clusters do not have to follow a "pattern" in a strict sense (e.g. neither exact symbol names nor their strict order have to match exactly), but ACL2(ml) considers structures and background information found in the library; and,
- working with potentially huge sets of ACL2 objects, ACL2(ml) makes its own intelligent discrimination of more significant and less significant patterns, based on cluster statistics.

To conclude this section, it is worth mentioning that ACL2(ml) can also work in a goal-directed mode and discover only clusters of functions that are similar to a given function f. This can speed-up the proof development in two different ways. Clustering will provide definitions of functions similar to f; hence, the proofs of the theorems involving those functions may follow similar patterns, and may need similar auxiliary lemmas. Clustering can also discover that a newly defined function f was previously defined; in that case, the user can use the existing library definition and all its background theory instead of defining it from scratch.

4 A Precondition Generator based on ACL2 Guards

In this section, we present the second contribution of this paper – an improvement of the lemma generation tool implemented in ACL2(ml). Given a target theorem, a source theorem, and a source lemma; the lemma analogy tool implemented in ACL2(ml) constructs a target lemma mutating the conclusion of the source lemma. Some of the mutated conjectures are true only under certain preconditions; however, these preconditions were not automatically generated in the initial ACL2(ml).

Example 4.1 *Consider that we are proving the equivalence between the recursive and tail-recursive versions of the Fibonacci function (theorem* `fib-fib-tail` *in Figure 2), and we request ACL2(ml) the generation of an auxiliary lemma for this target theorem. As a first step, ACL2(ml) finds the theorems that are similar to the theorem* `fib-fib-tail` *— as we have seen in Example 2.4, this theorem is similar to the equivalence theorem about the factorial function (*`fact-fact-tail`*). Subsequently, the lemma analogy tool will mutate the auxiliary lemma* `fact-helper-fact` *(used in the proof of* `fact-fact-tail`*) trying to generate a target lemma.*

Unfortunately, ACL2(ml) does not generate any valid suggestion – all of them are falsified during the counterexample checking phase. However, inspecting the non-valid conjectures, we find the following result: `(equal (helper-fib n j k) (+ (* (fib (- n 1)) j) (* (fib n) k))`*. This is the rewriting rule that is necessary to complete the proof of* `fib-fib-tail`*, but the necessary preconditions are missed. Note that using directly the preconditions of* `fact-helper-fact` *(as in Example 2.5) is not enough since we have an extra variable, and it is also necessary to impose the condition "n higher than* 0*".*

We have addressed this problem using the *guard* mechanism of ACL2 [19]. ACL2 is an untyped system; however, it provides a mechanism for restricting a function to a particular domain. Such restrictions are called *guards*, and they are used for two different aims: to increase the efficiency of the execution of ACL2 functions (proving that guards are satisfied allows ACL2 to directly use the underlying Common Lisp implementation to execute functions [12]), or as a specification device to characterise the kinds of inputs a function should have (the ACL2 value universe is divided into 14 pairwise-disjoint "primitive-types" and the ACL2 user can create new "types" by defining a predicate that recognises a subset of the ACL2 universe). We are interested in the latter.

The use of guards as specification device resembles the use of types in typed-languages. Then, we can use this "type" information to generate preconditions for the conjectures generated by ACL2(ml). However, guards are not mandatory when the user defines a function, and several functions do not provide explicitly their guards (in these functions, `T` is assigned as default guard value). To solve this problem, we have defined Algorithm 1 to compute the guard on a function f based on the guards of the functions called in the definition of f.

ACL2(ml) uses Algorithm 1 to compute the guard on a function, after it is defined in ACL2; in addition, the computed guard is stored for further use. The implementation of Algorithm 1 in ACL2(ml) invokes ACL2 with two different aims:

- obtain the guard information stored in ACL2: we use the keyword command `:args` to see the guard on a function; and,
- simplification of the final result: we use the `easy-simplify-term` function included in the book "tools/easy-simplify.lisp" to simplify the guard $\bar{g}$.

Example 4.2 *Let us use Algorithm 1 to obtain the guards of the function* `helper-fib` *(cf. Figure 2). This function calls the functions* `zp`*,* `equal`*,* `binary-+` *(*`-` *and* `+` *are macros that are expanded to the function* `binary-+`*) and* `helper-fib` *(note that* `helper-fib` *is a recursive function). The guards of these functions are given in Table 1. After combining and simplifying these guards, the result is:* `(and (integerp n) (not (< n 0)) (acl2-numberp j) (acl2-numberp k))`.

Given a conjecture c constructed by the lemma analogy tool of ACL2(ml), the guards of the functions invoked in c can be used to generate the preconditions of c, see Algorithm 2.

ALGORITHM 1: Recurrent Guard Generation

Input: A function f.
Output: The guard $\bar{g}$ on f.

ask ACL2 the stored guard $\bar{g}$ on f;
if $\bar{g} \neq T$ **then**
  return $\bar{g}$.
else
  $\bar{g} \leftarrow$ T;
  expand the macros in the definition of f;
  for each *function call* $(f_i\ t_i^1\ \ldots\ t_i^{n_i})$ *included in the expanded definition of* f **do**
    if *guard* g_i *on* f_i *was memoised* (f_i, g_i) **then**
      $\bar{g}_i \leftarrow g_i$ where its parameters are replaced with $t_i^1\ \ldots\ t_i^{n_i}$;
      $\bar{g} \leftarrow (\bar{g}$ and $\bar{g}_i)$;
    else
      ask ACL2 the stored guard g_i on the function f_i;
      case $g_i \neq T$:
        memoise (f_i, g_i) for further use;
        $\bar{g}_i \leftarrow g_i$ where its parameters are replaced with $t_i^1\ \ldots\ t_i^{n_i}$;
        $\bar{g} \leftarrow (\bar{g}$ and $\bar{g}_i)$;
      endsw
      case $g_i = T$ *and either* f_i *is an ACL2 built-in function or* $f_i = f$:
        memoise $(f_i,$ T$)$ for further use;
      endsw
      otherwise
        compute guard g_i on f_i using Algorithm 1;
        memoise (f_i, g_i) for further use;
        $\bar{g}_i \leftarrow g_i$ where its parameters are replaced with $t_i^1\ \ldots\ t_i^{n_i}$;
        $\bar{g} \leftarrow (\bar{g}$ and $\bar{g}_i)$;
      endsw
      return *simplification of* $\bar{g}$.
    end
  end
end

Function call	Function with formals	Guard	Guard with arguments
(zp n)	(zp x)	(and (integerp x) (<= x 0))	(and (integerp n) (<= n 0))
(equal n 1)	(equal x y)	T	T
(binary-+ n -1)	(binary-+ x y)	(and (acl2-numberp x) (acl2-numberp y))	(and (acl2-numberp n) (acl2-numberp -1))
(binary-+ j k)	(binary-+ x y)	(and (acl2-numberp x) (acl2-numberp y))	(and (acl2-numberp j) (acl2-numberp k))
(helper-fib (- n 1) k (+ j k))	(helper-fib n j k)	T	T

Table 1: *Guards of the functions used in the definition of* `helper-fib`.

ALGORITHM 2: Generation of Preconditions

Input: The conclusion of a conjecture c.

Output: Precondition p of c generated from the guards of the functions used in c.

$p \leftarrow$ `T`;
expand the macros in the conjecture c;
for each *function call* $(f_i\ t_i^1\ \ldots\ t_i^{n_i})$ *included in the expanded conjecture* c **do**
  use Algorithm 1 to compute the guard g_i on f_i;
  $\bar{g}_i \leftarrow g_i$ where its parameters are replaced with $t_i^1\ \ldots\ t_i^{n_i}$;
  $p \leftarrow (p$ and $\bar{g}_i)$;
end
return *simplification of p.*

Example 4.3 *Let us see the preconditions that are generated using Algorithm 2 for the conjecture* `(equal (helper-fib n j k) (+ (* (fib (- n 1)) j) (* (fib n) k)))` *from Example 4.1. The guards for the different functions included in this conjecture are given in Table 2. After combining and simplifying the guards, the generated precondition is:*

```
(and (integerp n) (not (< n 0))
     (acl2-numberp j) (acl2-numberp k)
     (not (< (+ -1 n) 0)))
```

These are the necessary conditions to prove the given conjecture – provided that the arithmetic library is included in the development.

It is worth mentioning that the preconditions generated by ACL2(ml) go beyond type information (for instance, in Example 4.3 the generated preconditions impose that `n` must be a natural number higher than 0). Note also that the preconditions generated by the guards may not be enough to prove a conjecture, since the necessary preconditions could be more complex than the information included in the guards.

The generator of preconditions is internally called by the lemma analogy tool of ACL2(ml); additionally, the user can also invoke it to obtain the preconditions of his own conjectures. This functionality can be especially useful for novice users of ACL2, since they often assume that a conjecture is restricted to the domain of interest when specifying their conjectures. However, the ACL2 logic needs all assumptions to be given explicitly; hence, the generator of preconditions can help them to add the missing preconditions to their conjectures. Another application of this tool is the discovery of false conjectures: if the guards generated from the functions of a conjecture contradict themselves, the user is trying to prove a result that is either false or meaningless.

Example 4.4 *Consider the tail-recursive function to compute the length of a list:*

```
(defun lengthTail (lst res)
   (if (endp lst) res (lengthTail (cdr lst) (1+ res))))
```

If we try to prove the conjecture `(equal (+ res (length lst)) (lengthTail res lst))` *(where we have swapped the arguments of the function* `lengthTail`*); and invoke the generator of preconditions, the returned result is* `nil`*. This indicates that there is a contradiction in the guards of the functions: the guard generated from* `+` *is* `(acl2-numberp res)`*, and the guard generated from* `lengthTail` *is* `(consp res)` *– this means that* `res` *should belong to two disjoint "types"; hence, the*

Function call	Function with formals	Guards	Guards with arguments
(fib n)	(fib x)	(and (integerp x) (<= x 0))	(and (integerp n) (<= n 0))
(binary-* (fib n) k)	(binary-* x y)	(and (acl2-numberp x) (acl2-numberp y))	(and (acl2-numberp (fib n)) (acl2-numberp k))
(binary-+ n -1)	(binary-+ x y)	(and (acl2-numberp x) (acl2-numberp y))	(and (acl2-numberp n) (acl2-numberp -1))
(fib (- n 1))	(fib x)	(and (integerp x) (<= x 0))	(and (integerp (- n 1)) (<= (- n 1) 0))
(binary-* (fib (- n 1)) j)	(binary-+ x y)	(and (acl2-numberp x) (acl2-numberp y))	(and (acl2-numberp (fib (- n 1))) (acl2-numberp j))
(binary-+ (* (fib (- n 1)) j) (* (fib n) k))	(binary-+ x y)	(and (acl2-numberp x) (acl2-numberp y))	(and (acl2-numberp (* (fib (- n 1)) j)) (acl2-numberp (* (fib n) k)))
(helper-fib n j k)	(helper-fib n j k)	(and (integerp n) (not (< n 0)) (acl2-numberp j) (acl2-numberp k))	(and (integerp n) (not (< n 0)) (acl2-numberp j) (acl2-numberp k))

Table 2: *Guards of the functions used in the conjecture* `(equal (helper-fib n j k) (+ (* (fib (- n 1) j)) (* (fib n) k))`.

conjecture is false. If we fix the conjecture putting the arguments in the correct position, the generator of preconditions returns the necessary assumption, `(acl2-numberp res)`*, to prove the lemma.*

5 Related Work, Conclusions and Further Work

Related Work. ACL2(ml) combines statistical and symbolic machine-learning methods to guide the ACL2 user during the proof development. Statistical machine-learning methods can discover data regularities based on numeric proof features. Among other successful statistical tools is the method of statistical proof-premise selection [9, 10, 20, 24, 25]. Applied in several automated (first-order) theorem provers, this method provides statistical ratings of existing lemmas; and this information is used to make automated rewriting more efficient. There are several differences between the premise selection tools and ACL2(ml):

- Both the premise selection tools and ACL2(ml) use matrices to encode term trees, but the former uses sparse incidence matrices (capturing the occurrence of the symbols of a library in the term tree) and the latter uses a special feature matrix (cf. Figure 3) that captures the structure and dependencies of the function symbols occurring in the tree.
- The premise selection tools use supervised machine-learning techniques – in particular, a classifier is constructed for every lemma L of a given library, and such a classifier is trained to estimate the likelihood of using the lemma L during a proof. On the contrary, ACL2(ml) uses unsupervised clustering to discover families of similar theorems and definitions.
- The aim of the premise selection tools is to make automated rewriting more efficient – the methods applied in these tools could be used to improve the efficiency of ACL2's rewriter. Then, these tools are focused on helping the prover; on the contrary, ACL2(ml) was born as a hint provider to help the user in the discovery of similarities across libraries and the formulation of auxiliary results.

The task of formulating auxiliary lemmas can be undertaken either manually or using symbolic machine learning techniques. While statistical methods focus on extracting information from existing large theory libraries, symbolic methods are instead concerned with automating the discovery of lemmas in new theories [5, 6, 16, 17, 21, 22], while relying on existing proof strategies, e.g. proof-planning and rippling [1]. These methods can be slow on large inputs due to the increase in the search space. This problem is overcome in ACL2(ml) feeding the output of statistical machine-learning algorithms (families of similar theorems) to a mutation tool that uses this data for more efficient lemma discovery.

Conclusions and Further Work. In this paper, we have presented two extensions for ACL2(ml) that advance ACL2(ml) further in the direction of guiding the ACL2 user during the proof development. The

first extension allows the user to detect families of similar function definitions. This can be used to reduce the redundancies in ACL2 books, reuse previous developments, and find similar definitions that can be helpful for the user's current development. The second extension enhances the quality of the conjectures constructed by ACL2(ml), not only analogising the conclusions of theorems but also generating some preconditions using the guard mechanism of ACL2. This extension can also be invoked directly by the user to generate preconditions for his own conjectures.

ACL2(ml) is currently implemented as an extension for Emacs (one of the two main ACL2 development environments), we are planning to reimplement some of ACL2(ml) modules (namely, the generators of conjectures and preconditions) as ACL2 books to integrate them in other environments, or use them without running Emacs. We are also exploring different alternatives to improve the heuristics to generate auxiliary lemmas (for instance, using the information shown by ACL2 in failed proof-attempts), and studying how to generate new functions based on the statistical results obtained using clustering. In addition, we are interested in enhancing the generator of preconditions. In general, ACL2 lemmas should be stated as general as possible keeping the number of hypothesis to a minimum; however, the generator of preconditions can introduce unnecessary hypothesis in the conjectures constructed by ACL2(ml). The `remove-hyps` tool [26] was introduced to remove unnecessary hypothesis from theorems; and therefore it would be helpful to incorporate this tool into ACL2(ml).

Acknowledgements

We are grateful to the anonymous referees for their useful suggestions and comments; and J Moore for useful discussions about this work.

References

[1] D. Basin et al. (2005): *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press.

[2] C. Bishop (2006): *Pattern Recognition and Machine Learning*. Springer.

[3] B. Brock, M. Kaufmann & J S. Moore (1996): *ACL2 Theorems about Commercial Microprocessors*. In: *1st International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, *LNCS* 1166, pp. 275–293, doi:10.1007/BFb0031816.

[4] H. Chamarthi, P. Dillinger, M. Kaufmann & P. Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In: *10th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'11)*, *EPTCS* 70, pp. 4–19, doi:10.4204/EPTCS.70.1.

[5] K. Claessen et al. (2013): *Automating Inductive Proofs using Theory Exploration*. In: *24th International Conference on Automated Deduction (CADE-24)*, *LNCS* 7898, pp. 392–406, doi:10.1007/978-3-642-38574-2_27.

[6] S. Colton (2002): *The HR Program for Theorem Generation*. In: *18th International Conference on Automated Deduction (CADE-18)*, *LNCS* 2392, pp. 285–289, doi:10.1007/3-540-45620-1_24.

[7] J. Davis et al.: *ACL2 Community Books*. `https://code.google.com/p/acl2-books/`.

[8] J. Davis et al. (2009–2014): *XDOC Documentation System for ACL2*. `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/`.

[9] J. Denzinger, M. Fuchs, C. Goller & S. Schulz (1999): *Learning from Previous Proof Experience: A Survey*. Technical Report, Technische Universitat Munchen.

[10] J. Denzinger & S. Schulz (2000): *Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts*. Information and Computation 162(1-2), pp. 59–79, doi:10.1006/inco.1999.2857.

[11] C. Eastlund, D. Vaillancourt & M. Felleisen (2007): *ACL2 for Freshmen: First Experiences*. In: 7th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'07), ACM Press, pp. 200–211.

[12] D. Greve et al. (2008): *Efficient Execution in an Automated Reasoning Environment*. Journal of Functional Programming 18(1), pp. 15–46, doi:10.1017/S0956796807006338.

[13] M. Hall et al. (2009): *The WEKA Data Mining Software: An Update*. SIGKDD Explorations 11(1), pp. 10–18, doi:10.1145/1656274.1656278.

[14] J. Heras & E. Komendantskaya (2013–2014): *ACL2(ml): downloadable programs, manual, examples*. `http://staff.computing.dundee.ac.uk/katya/ACL2ml`.

[15] J. Heras, E. Komendantskaya, M. Johansson & E. Maclean (2013): *Proof-Pattern Recognition and Lemma Discovery in ACL2*. In: 19th Logic for Programming Artificial Intelligence and Reasoning (LPAR-19), LNCS 8312, pp. 389–406, doi:10.1007/978-3-642-45221-5_27.

[16] S. Hetzl, A. Leitsch & D. Weller (2012): *Towards Algorithm Cut-Introduction*. In: 18th Logic for Programming Artificial Intelligence and Reasoning (LPAR-18), LNCS 7180, pp. 228–242, doi:10.1007/978-3-642-28717-6_19.

[17] M. Johansson, L. Dixon & A. Bundy (2011): *Conjecture Synthesis for Inductive Theories*. Journal of Automated Reasoning 47(3), pp. 251–289, doi:10.1007/s10817-010-9193-y.

[18] M. Kaufmann, P. Manolios & J S. Moore, editors (2000): *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0.

[19] M. Kaufmann, P. Manolios & J S. Moore, editors (2000): *Computer-Aided Reasoning: An approach*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-4449-4.

[20] D. Kühlwein, T. van Laarhoven, E. Tsivtsivadze, J. Urban & T. Heskes (2012): *Overview and Evaluation of Premise Selection Techniques for Large Theory Mathematics*. In: 6th International Joint Conference on Automated Reasoning (IJCAR'12), LNCS 7364, pp. 378–392, doi:10.1007/978-3-642-31365-3_30.

[21] R. L. McCasland, A. Bundy & P. F. Smith (2006): *Ascertaining Mathematical Theorems*. In: 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005), ENTCS 151, pp. 21–38, doi:10.1016/j.entcs.2005.11.021.

[22] O. Montano-Rivas, R. Mccasland, L. Dixon & A. Bundy (2012): *Scheme-based theorem discovery and concept invention*. Expert Systems with Applications 39(2), pp. 1637–1646, doi:10.1016/j.eswa.2011.06.055.

[23] R. Page (2007): *Engineering Software Correctness*. Journal of Functional Programming 17(6), pp. 675–686, doi:10.1017/S095679680700634X.

[24] E. Tsivtsivadze, J. Urban, H. Geuvers & T. Heskes (2011): *Semantic Graph Kernels for Automated Reasoning*. In: SIAM Conference on Data Mining (SDM'11), SIAM / Omnipress, pp. 795–803, doi:10.1137/1.9781611972818.68.

[25] J. Urban, G. Sutcliffe, P. Pudlák & J. Vyskocil (2008): *MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance*. In: 4th International Joint Conference on Automated Reasoning (IJCAR'08), LNCS 5195, pp. 441–456, doi:10.1007/978-3-540-71070-7_37.

[26] N. Wetzler & M. Kaufmann (2014): *Remove-hyps and Writing Utilities with Make-event*. ACL2 Theorem Proving Seminar. University of Texas.