# Building Lemmas Using Examples

Gabriel Infante-Lopez
Universidad Nacional de Córdoba
Consejo Nacional de Investigaciones Científicas y Técnicas
Córdoba, Argentina
gabriel@famaf.unc.edu.ar

## ABSTRACT

We present a heuristic for automated lemma discovery that generates lemmas that might help ACL2 in proving theorems like $\forall x : t_1(x) = t_2(x)$. This heuristic exploits manually created examples of $x$. These examples are used to produce ground terms $t'_1$ and $t'_2$, for which semantical models are built. In order to generate useful intermediate lemmas, we search for a specific pattern in these two models. The lemmas suggested by our heuristic are of the form $\forall x : h(g_1(x)) = h(f_1(x))$. A lemma is suggested if and only if $t'_1$ and $t'_2$ can be rewritten as terms containing subterms $h(g_1(a))$ and $h(f_1(a))$ respectively, such that $h(g_1(a)) = h(f_1(a))$ but $g_1(a) \neq f_1(a)$. We explain how to search for these patterns and how to build lemmas from a collection of ground equalities.

## Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Deduction and Theorem Proving

## General Terms

Algorithms, Theory

## Keywords

automated theorem proving, lemma discovery, program synthesis

## 1. INTRODUCTION

The ACL2 theorem prover can easily prove that two functions $f(x)$ and $g(x)$ compute the same result for a particular value $a$: It compares the results of evaluating $f(a)$ and $g(a)$. In contrast, showing that both functions compute the same value for all possible values of $x$ is a completely different task. For this purpose, ACL2 may need intermediate lemmas. The need for such intermediate lemmas is one of the main reasons for user interaction/guidance while ACL2 is attempting to prove something. Various strategies have been developed for building intermediate lemmas, e.g., [1, 2]. Most of them have in common that they use symbolic strategies that are mainly based upon rewriting systems, but none take into consideration how terms are effectively reduced or computed. In other words, none of them compares the process that makes $f(x)$ equal to $g(x)$. This is because, in principle, it is not possible to observe the computations of $f(x)$ and $g(x)$ as both terms have free variables.

In order to elaborate a strategy based on observation of computational models for suggesting lemmas, two main steps have to be fulfilled. First, we need to define the nature of the object that codifies the computation of a term. This object should represent a kind of semantic model and it is not necessarily a trace on a semantic machine; we picture it as an object that keeps track of the intermediate evaluation of subterms. Since the goal of the approach is to compare computations that produce the same results but that are carried out in different ways, we need this semantic object to be easily computed and compared. It is important to note that the semantic object that we want to define is not modelling the final result of the computation, instead, its focus in on the process that produces this final result. Second, we need to define strategies that construct examples of ground terms $a$ such that the computation of $f(a)$ and $g(a)$ can be observed. Ideally, these strategies should suggest those examples $a$ that best reveal insightful information.

In this paper we present work in progress focused on the first step. We introduce a semantic model, called *evaluation graphs*, that models ground terms expressed in `lisp`. An evaluation graph contains all possible ways to expand function calls together with all possible partial evaluations of all possible subterms. All subterms are evaluated in a lisp interpreter and their evaluation values are used to compare evaluation graphs coming from terms that are believed to compute the same value in different ways. The comparison of the two graphs generates lemmas that might help to determine the equivalence of the two terms. In terms of rewriting systems, our approach can be summarized as follows. We present a lemma discovery heuristic that generates lemmas by comparing the computational model of two closed terms $f(a)$ and $g(a)$. The heuristic rewrites $f$ and $g$ into two terms $f'$ and $g'$ that share the same set of functions. Then, it compares the two terms searching for subterms $h(f_1(a_i))$ in $f'$ and $h(g_1(a_i))$ in $g'$ such that $h(f_1(a_i))$ and $h(g_1(a_i))$ evaluate to the same value but $f_1(a_i)$ does not evaluate to the same value as $g_1(a_i)$. From equalities between ground terms we generate the lemmas that are suggested to ACL2.

The rest of the paper is organized as follows. Section 2 presents evaluation graphs. Section 3 explains how to use them for lemma generation. Section 4 shows an example where we apply our strategy. Finally, Section 5 concludes the paper and states our plans for future research.

## 2. COMPUTATIONAL MODEL

Our approach is based on the idea of comparing semantic models for ground terms. We start by defining what we understand as semantic models. Given a term $t$, its computation is modelled by *evaluation graphs*. An evaluation graph $G_t$ for a term $t$ is a labelled directed graph that is built using term $t$ by progressively evaluating parts of $t$ in a lisp interpreter. Vertices in $G_t$ are called *evaluation vertices*, and each of them consists of tuples $\langle r, e, R\rangle$ where $r$ is a tree-term, $e$ is the evaluation of $r$ – equivalent to computing `(eval r)` (not necessarily simplified) in a lisp interpreter – and $R$ is a set of ground terms.

Terms are expressed as trees. A *term-tree* for a term $t$ is the syntactic tree that corresponds to term $t$. The tree in Figure 1 (b) is the tree-term that corresponds to the term `(tail '(1 2 3))`. We use tree-terms and terms indistinguishably in the rest of the paper. We refer to nodes in tree-terms as *nodes*, and to nodes in an evaluation graph as *vertices*.

$G_t$ contains a special vertex, called the *root vertex*, that represents the term $t$ and is given by $\langle t, e, \emptyset\rangle$, with $e$ equal to `(eval t)`, i.e., the value that results from evaluating term $t$. Note that this vertex is well defined: since $t$ is a closed term, `(eval t)` becomes computable. The set of vertices of an evaluation graph is constructed progressively by applying two different operations, namely *term extraction* and *term expansion*, to vertices that have already been constructed. Initially, the set of vertices contains only the root vertex. Since every vertex is built from an existing one, it is easy to show that if a vertex $v$ belongs to the set of vertices, then there exist intermediate vertices $v_1, \ldots, v_n$ such that $v_1$ is the root vertex and $v_j$ is the result of applying one of the two operations to $v_{j-1}$ for all $j$.

These two operations take two arguments: a vertex $\langle t, e, R\rangle$ and a *path definition*. A path definition is a way to select a node in a tree-term. Paths are described by possibly empty sequences of integers. As such, the empty string corresponds to the root node of a tree while a path $x : xs$ corresponds to the node indicated by path $xs$ in the $x$-th subtree of $t$. If $t$ is a term, and $\alpha$ is a path, then $t_\alpha$ is the subtree that hangs from the node selected by $\alpha$. For example, if $t$ is `(if (atom '(1 2 3)) '(1 2 3 ) (tail (cdr '(1 2 3))))`, its syntactic tree is shown in Figure 1 (d), $t_2$ is the subterm `(tail (cdr '(1 2 3)))` whose tree is picture in Figure 1 (e).

The set of edges of an evaluation graph is used to help with keeping track of how all vertices were created: if vertex $v_j$ was obtained from vertex $v_i$ by applying one of the two operations, then there is a directed edge from $v_i$ to $v_j$. Moreover, this arc is labelled with both the name of the operation and the path that was given as an argument.

Both operations construct a new vertex by modifying existing ones; the operation either expands a function name by introducing its definitions or extracts a subterm. In what follows, we define both.

### 2.1 Function Expansion

The Function-Expansion operation can be defined in terms of 3 different procedures: The first procedure consists of inserting the tree that represents the body of a function definition into a tree term. Formally, let $v = \langle t, e, R\rangle$ and $\alpha$ be, respectively, the tree-term and the path that are given as arguments. Suppose that $\alpha$ points to a node in $t$ that is labelled with function name $F$. Let $T_F$ be the tree-term that represents the body of $F$. The first step of the operation creates a new tree-term $t'$ by replacing $t_\alpha$ by $T_F$ and by replacing $T_F$ arguments by the corresponding daughters of $t_\alpha$. For example, Figure 1 (c) represents the body definition of the function `tail`. The first step, when applied to the Figure 1 (b) with the empty paths $\epsilon$ as argument, produces the tree in Figure 1 (d).

The second procedure traverses the tree $t'$ generated by the first procedure, in this case looking for nodes that are labelled with the `if` function. For every if-node it encounters, their conditionals are evaluated and the if-nodes are replaced by its second or third child depending on the result of the evaluation of the condition. For example, the tree in Figure 1 (d) is rewritten as the tree in Figure 1 (e) because the term `(atom '(1 2 3))` returns `nil`.

Finally, the third procedure collects a set of equality terms (terms like `(equal a b)`). Each of them is defined using the terms that appeared as first arguments of if-nodes and that were deleted in the second procedure: If a term `u` appeared as the first argument of an `if` node, then the term `(equal (u nil)` or `(not (equal u nil))` is collected depending on whether `u` evaluted to false of true respectively.

The operation of Function Expansion is defined using these 3 procedures as follows. Suppose the Function Expansion operation is called with arguments $\langle t, e, R\rangle$ and path $\alpha$. The result of the operation is that a new vertex $\langle t', e', R'\rangle$ and a new arc $l$ are added to the graph, where $t'$ is the result of applying the first two procedures to $t$ and $\alpha$. $e'$ is equal to $e$ given that inserting a function definition does not change the evaluation value and $R'$ is the union of $R$ and the output of the third procedure. Finally, the arc $l = (\langle t, e, R\rangle, \langle t', e', R'\rangle)$ connects the vertex given as argument and the new vertex and it is labelled with "Function Expansion: $\alpha$".

### 2.2 Term Extraction

The Term Extraction operation also takes as arguments a vertex and a path. Intuitively, it creates a new vertex by extracting part of the information that is contained in the vertex given as argument. Formally, suppose that the Term Extraction operation is called with arguments $\langle t, e, R\rangle$ and $\alpha$. The new vertex $\langle t', e', R'\rangle$ is defined as follows. $t'$ is $t_\alpha$, $e'$ is the result of evaluating $t_\alpha$ in a lisp interpreter, and $R \subseteq R'$ is defined as all if-conditions that were created while reducing both nodes along the path $\alpha$ and nodes in $t_\alpha$. For example, if the Term Extraction operation is applied to arguments $\langle t, e, R\rangle$, and $\alpha = 0$ where $t$ is the tree pictured in Figure 1 (e), $e$ equals to 3, and $R$ equals to $\{$`(equal (atom '(1 2 3)) nil)`$\}$, the result is a new vertex $\langle t, e, R\rangle$, where $t =$ `(cdr '(1 2 3))`, $e =$ `(2 3)` and $R' = \{$`(equal (atom '(1 2 3)) nil)`$\}$.

```
(defun tail (x)
  (if (atom x)
      x
      (tail (cdr x))))
```

(a)

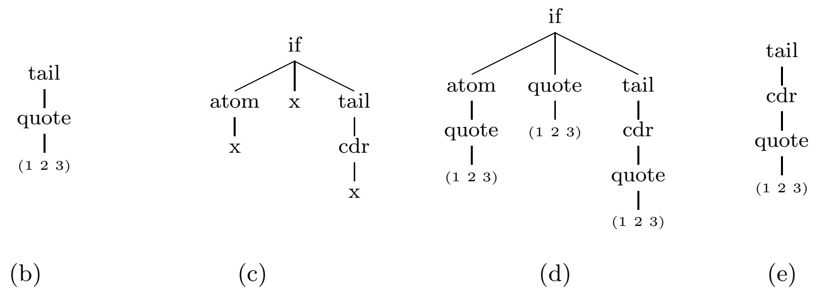

**Figure 1: (a) Definition of `tail`. (b) tree representing term `(tail '(1 2 3))`. (c) tree representing the body of function `tail`. (d) tree in (a) where the root has been expanded. (e) tree resulting from removing if-conditions from the tree in (d).**

The set $R$ keeps track of all the terms in the if-nodes that have been evaluated during the second procedure of the Function Expansion operation. Whenever a subtree $t'$ is extracted from a tree $t$, all the if-conditions that were used for reducing $t$ have to be added to $R$. As we will see in the following section, $R$ is used for lemma generation.

Clearly, the number of vertices contained in an evaluation graph depends on the number of functions that are expanded and the number of subterms that are extracted. If all functions that are called in the term of the root vertex terminate, then we can show that its evaluation graph is finite. As we said before, our heuristic compares two evaluation graphs coming from two different terms. In order to make these two graphs comparable, we require that they have vertices whose tree-terms share the same functions being called, that is, both tree-terms are expressed using the same functions. It has to be possible to rewrite both terms until both of them are expressed in terms of basic functions. In order to guide the rewriting procedure, we introduce the concept of *set of basic functions*. Functions that are in the set of basic functions are not expanded. In contrast, all nodes that are labeled with functions that do not belong to this set are extracted or expanded. An evaluation graph is constructed by applying both operations to all vertices whose terms do not have a node labelled with a basic function. It is important for our purposes that the operator `quote` belongs to the set of basic functions.

## 3. FINDING LEMMAS

In this section we describe the use of evaluation graphs for lemma generation. Our main aim is to generate intermediate lemmas that might help proving lemmas like `(defthm lemma-to-prove (equal (f x) (g x)))` which involves the equality between two functions `f` and `g`. Since our model requires terms to be grounded, we manually define the set of ground terms that are used as arguments of $f$ and $g$.

In order to find lemmas, we keep track of how the two evaluation graphs compute their results. In order to relate the two computation models, we define a relation $\sim$ between vertices in the two graphs. As we will see below, two vertices are related if, first, their evaluation value is the same and, second, it is computed in the right place of the computation. Let us make this idea more formal. Let $G_{f(a)} = \langle V_{f(a)}, E_{f(a)} \rangle$, and $G_{g(a)} = \langle V_{g(a)}, E_{g(a)} \rangle$ be evaluation graphs that correspond to terms $f(a)$ and $g(a)$ respectively. Let $\sim$ be a relation between the set of vertices $V_{f(a)}$ and $V_{g(a)}$ defined as follows: A vertex $v_{g(a)}$ in $V_{g(a)}$ is related to a vertex $v_{f(a)}$ in $V_{f(a)}$ if, first, both vertices share the same evaluation value and, second, *one* of the following items is satisfied:

1. $v_{g(a)}$ and $v_{f(a)}$ are the root vertices of graphs $G_{g(a)}$ and $G_{f(a)}$ respectively.
2. There exist vertices $v'_{g(a)}$, $v'_{f(a)}$ and edges $(v'_{g(a)}, v_{g(a)})$, $(v'_{f(a)}, v_{f(a)})$ both labelled with "Function-Expansion: $\alpha$" such that $v'_{f(a)}$ is related to $v'_{g(a)}$.
3. There exist vertices $v'_{g(a)}$, $v'_{f(a)}$ and edges $(v'_{g(a)}, v_{g(a)})$, $(v'_{f(a)}, v_{f(a)})$, labelled with "Term-Extraction: $\alpha$'," such that $\alpha$ has length equal to 1, $v'_{f(a)}$ is related to $v'_{g(a)}$, both root nodes of their tree-terms are labelled with the same function name and both root nodes have the same number of descendents.

In order to build lemmas using relation $\sim$, we first need to introduce some auxiliary concepts. Since our algorithm builds evaluation graphs using terms $f(a)$ and $g(a)$, a ground term $a$ has replaced variables in both terms $f$ and $g$. For any vertex in evaluation graphs, it is still possible to replace original occurrences of $a$ by a free variable $x$. This is the case because subtrees labelled with `quote` are not extracted nor rewritten, as a consequence of the function `quote` belonging to the set of basic functions. It is not possible to find half of a term $a$ in one subtree and the other half in another subtree. Replacing back argument $a$ by free variables produces terms with free variables that are used for building lemmas. Let us call $lift_a$ the operation of replacing appropriate occurrences of $a$ by a fresh free variable $x$. Specifically, if $f(a)$ is a ground term, then $lift_a(f(a))$ returns a term with a free variable for every occurrence of $a$, which was initially a free variable. The operation $lift_a$ applied to a set of terms $R$ returns $\bigwedge_{f \in R} lift_a(f)$.

The relation $\sim$ expresses many different formulae between terms. Let $\langle t, e, R \rangle$ and $\langle t', e', R' \rangle$ be two vertices in $G_{f(a)}$ and $G_{g(a)}$ respectively such that $\langle t, e, R \rangle \sim \langle t', e', R' \rangle$. If it is true that $f(x) = g(x)$ for a ground term $a$, then we speculate that it is true that $lift_a(R) \wedge lift_a(R') \Rightarrow lift_a(t) = lift_a(t')$. In particular, if $\langle t, e, \emptyset \rangle$ and $\langle t', e', \emptyset \rangle$ are root vertices of evaluation graphs $G_{f(a)}$ and $G_{g(a)}$ respectively, then, if they are related, we speculate that the formula $f(x) = g(x)$ holds.

Intuitively, the $\sim$ relation encodes all possible equalities between subterms of $f$ and $g$. The relation deconstructs the equality between $f(x)$ and $g(x)$ in terms of more basic equalities. Note that $\sim$ is constructed by a heavy use of the evaluation of subterms.

## 3.1 Searching For Patterns

The relation $\sim$ encodes many different lemmas, but not all of them are of interest. In our case we search for a particular pattern in the relation $\sim$. We seek vertices $v$ in $G$ and $v'$ in $G'$ that satisfy the following requirements: (a) $v \sim v'$ (b) their corresponding tree terms share the same label at their root name, and (c) there exist one vertex $w$, and an arc $(v, w)$ (respectively, $(v', w)$ ) labelled with "extract:$\alpha$" in $G$ (respectively $G'$); $|\alpha| = 1$ such that $w$ is not $\sim$-related to any neighbor of $v'$ (respectively $v$) connected to $v'$ (respectively $v$) with an edge with labeled "extract:$\alpha$".

In other words, we look for subterms $h(g_1(a), \ldots, g_n(a))$ and $h(f_1(a), \ldots, f_m(a))$ in $G$ and $G'$ respectively such that both terms evaluate to the same value but $m \neq n$ or there is an argument $i$ such that $g_i(a) \neq f_i(a)$. For each pair of vertices $v$ and $v'$ that satisfies this pattern we propose as a new lemma the one that is encoded in the relation between these two vertices.

## 3.2 Filtering Formulas

While comparing two graphs, many lemmas are generated. Some of them are too specific to the grounding that is used for building $\sim$. In order to reduce the number of lemmas that this procedure produces, we filter out some of the generated lemmas. So far, we have implemented two different filtering mechanisms. One of them generates different set of lemmas using different groundings and it reports only those lemmas that are common to all sets. The other mechanism uses a complexity measure defined over lemmas and reports those lemmas having a small value for it.

## 4. EXAMPLE

In this section we show our heuristic in action. We use an example from [4] where two different functions compute the same value and where we find it necessary to prove the equality between the two. We build the two evaluation graphs and we search for the patterns described in the previous sections. We use the following functions in our example:

```
(defun flatten (x)
  (cond ((atom x) (list x))
  (t (append  (flatten (car x)) (flatten (cdr x))))))
(defun gopher (x)
  (if (or (atom x)  (atom (car x))) x
    (gopher (cons (caar x) (cons (cdar x) (cdr x))))))
(defun samefringe (x y)
  (if (or (atom x) (atom y)) (equal x y)
    (and (equal (car (gopher x)) (car (gopher y)))
    (samefringe (cdr (gopher x)) (cdr (gopher y))))))
```

Because of the Lisp interpreter we use, the function `equal` that is used in our example is treated as a non-basic function. That is, we had to define it in terms of the function `eql` which is a basic functions. Finally, the following states the equality we want to prove.

```
(defthm correctness-of-samefringe
  (equal (samefringe x y)(equal (flatten x)(flatten y)))
```

In order to generate lemmas that help prove this equality, we built several evaluation graphs for terms `(samefringe x y)` and `(equal (flatten x) (flatten y))` using different values of `x` and `y`; for each grounding we build evaluation graphs, and finally we generate the lemmas for vertices that satisfy the pattern presented in the previous section.

We tried the following pairs of ground terms for variables `(X, Y)`: `('3, '3)`, `('(1 2), '( 1 2))` and `('((1 2) . 3), '(1 . (2 3)))`. These terms were chosen arbitrarily and were generated by hand. The first grounding does not produce any lemma: the pattern we are interested in does not appear in the comparison between the two graphs. All others produce a number of lemmas that grows quite fast with the size of the term and they all have one lemma in common:

```
(implies (and (not (atom x)) (consp (flatten x)))
  (eql (car (gopher x)) (car (flatten x))))
```

removing trivial clauses our lemma can be rewritten as

```
(implies (consp  x )
  (eql (car  (gopher  x))  (car (flatten  x ))))
```

This lemma is one of the lemmas required by ACL2 to prove the equality of `correctness-of-samefringe`

## 5. DISCUSSION AND FUTURE WORK

We have presented a heuristic for discovering lemmas that is based on a computational model that evaluates and rewrites terms. Our computational model is based on graphs that keep track of all possible ways to rewrite a function. We showed an example where our heuristics helps finding an intermediate lemma. Still, one example is not enough to ensure that our heuristic helps. It is our intention to build a test set where our ideas can be tested.

We believe that the lemmas generated by this procedure can be used in two different ways: as a tool to help ACL2 find intermediate lemmas, or as a tool to help ACL2 select which books to load. For the second, the book that best fits the set of lemmas that our tool generated should be loaded. We have also developed a graphical tool that shows evaluation graphs; we believe that such a tool can help humans interfacing with ACL2 because it provides a compact, traversable, and graphical representation of a computation. The tool can also be extended to show the relation between two evaluation graphs.

As drawbacks, our approach requires some meaningful examples of ground terms to produce lemmas. How to automatically generate non-trivial examples is still an open issue that requires further research. Moreover, our approach lacks of strategies to generate lemmas that can help proving equalities by induction as in [3].

## 6. REFERENCES

[1] S. Colton, A. Bundy, and T. Walsh. Automatic concept formation in pure mathematics. In *IJCAI*, 1999.

[2] M. Demba and K. Bsaïes. Appropriate lemmae discovery. *Inf. Sci.*, 163(4):221–237, 2004.

[3] D. Kapur and M. Subramaniam. Lemma discovery in automating induction. *LNCS*, 1104, 1996.

[4] M. Kaufmann, P. Manolios, and J. Moore. *Computer–Aided Reasoning: An Approach.* Kluwer, 2000.