

Overview of the ACL2-Language and System

Sven Wille

18.09.2020

Abstract

ACL2 is a industry strength, first order theorem prover, focusing on automated proving [7] . It is rather unique in the way the language and proofs work compared to other more commonly used provers. The design goal of ACL2 is to give the user the ability to formalize and proof large programs and systems.

This work will give an overview of the ACL2 language and system and is split into two parts. The first part gives a more abstract overview of ALC2 and puts it into perspective with more prominent systems. The second part introduces concrete ACL2 by working through several examples.

1 Introduction

Modern software tries to ensure reliability by testing extensively. Although proving would guarantee that a program behaves according to its specification proving is rarely used in real world software development due to its complexity and amount of work required. To mitigate these shortcomings there is a constant effort to automate the process of proving and thus make it more accessible and less laborious. A system that has a very strong emphasis on automation is ACL2, an industry strength proof-assistant that has been used for such things as [7] :

- certifying several microcode programs for the Motorola CAP digital signal processor
- verifying the compliance of the AMD Athlon’s (TM) elementary floating point operations with their IEEE 754 specifications
- verifying the floating point divide and square root on the IBM Power 4 microprocessor

2 The ACL2 language and system

ACL2, developed at the university of texas, stands for “A Computational Logic for Applicative Common Lisp”, hence the “2” (the predecessor is called

nqthm). More concretely, ACL2 is a first order purely functional programming language, a mathematical logic to reason about programs and a proof assistant [5] .

The Language is designed as a small subset of common lisp. Things that are missing from common lisp include the class system, higher order functions and side effects, among other things.

Functions in ACL2 have to be total, well typed and terminating. As ACL2 is a first-order system, there are also no higher order theorems.

Like common lisp, ACL2's typing system is primarily dynamic with the exception of when a functions is defined. Types in ACL2 are implemented using boolean predicates. This results in ACL2 not having proper pattern-matching. Instead pattern-matching is realized by using boolean expressions.

The underlying logic of ACL2 is quantifier-less FOL plus equality. The consequence is that free variables in theorems are implicitly all-quantified.

3 Related work

Most proof assistants use proof-automation to greater or lesser extend. Two of the most popular proof assistants that use automation are Coq and Isabelle. Both systems vary quite heavily from ACL2 and to lesser degree from each other.

3.1 Coq

Coq is a dependently typed proof assistant and programming language developed by Inria [1] [11]. Proofs in Coq are mainly carried out by telling the proof assistant in a step-by-step fashion what it is supposed to do. Nonetheless Coq offers facilities to automate parts of a proof or the proof all together. The basic builtin general purpose automatic proof methods (auto, eauto, etc.) in Coq are fairly weak or require a lot of manual adjustment [13]. As an example, the often used auto tactic on its own barely can proof anything that is not considered "trivial". It needs at least a database that contains hints (i.e. already proven theorems). But even then its use is limited.

There are several stronger specialized automatic proof methods that work quite well for their domain, like FOL or linear arithmetic, but due to their nature can only be used for very small subset of theorems.

A weak point of the builtin automation in Coq is that there are no proof traces or the traces very uninformative [13]. Most traces only show the steps taken during a proof attempt but not the proof state after every or some steps itself.

Coq's strength is the ability the create extensions to the builtin automation via the use of Ltac [2]. Ltac is a sub-language that is build into the Coq

system with which complex recursive tactics can be created that itself use other automation such as the aforementioned auto tactic to enhance the builtin proof-automation.

3.2 Isabelle/HOL

Isabelle, developed by several institutions including the university of Cambridge and Munich, is a logical framework in which different object logics can be implemented [9] [10]. The most advanced of these is the HOL-Implementation which has a wide range of proof-automation that includes several general purpose, arithmetic and logic proof methods. Its lack of dependent types makes Isabelle/HOL less expressive than Coq, though. But for the most part, the Isabelle/HOL language is expressive enough to formalize and proof virtually all interesting theorems one way or another.

Isabelle has strong proof automation including the use of first order provers such as Z3 or vampire. It has several general purpose automatic proof methods that can handle non trivial proofs to varying degree. The system, however, still needs the user to do some manual adjustment when using more powerful automation especially with regards to induction.

A special mentioning deserves the so called “sledgehammer” in Isabelle. This proof automation takes all other automatic proof methods and a database of already proven theorems and tries to combine several proof methods to solve the goal. When successful it presents the user with the instructions it used to solve the goal.

Only the weaker proof automation has some tracing capabilities which give the user information about the steps taken and some information about the proof states.

Like Coq, Isabelle has a sub-language, called Eisbach, that is used to extend its builtin proof-mechanisms [3].

3.3 Tabular overview of Coq, Isabelle/HOL and ACL2

This sections gives a rough tabular comparison of ACL2 with Coq and Isabelle including points that aren’t touched upon in this work [8] [14] [6]

type checking	Coq	Isabelle	ACL2
type system	static	static	dynamic
syntax	full dependent types	HOL	predicates
higher order functions	Coq/ML	Isabelle/Haskell	Lisp
real numbers	yes	yes	partial
interactive proving	std. lib	std. lib	modified version of ACL2
underlying logic	very expressive	very expressive	limited
partial functions	CoC	HOL	quantifier-less FOL + equality
logical framework	no	yes	no
strong proof automation	no	yes	no
extensible by the user	yes	yes	yes
large standard library	yes	yes	no
			yes

3.4 Comparing key aspects of ACL2 with Coq and Isabelle

ACL2 is, compared to Coq and Isabelle, not very expressive. It lacks features that are considered basic for most functional languages including Coq and Isabelle. These include higher order functions and pattern matching. This makes theorems and functions sometimes rather verbose in ACL2 since some functionality has to be mimicked.

With regards to typing, ACL2 can express most complex types with its predicate mechanism. In that regard its expressiveness is on par with Coq and is more expressive than Isabelle (ignoring higher order statements).

Proving in ACL2 is done mainly automatically. Compared with Coq and Isabelle it has arguably the most intuitive proof automation due to its re-traceability (as will be shown in the examples below) and potentially the strongest. Unlike Coq and Isabelle, ACL2's automation can handle induction out of the box. In Coq or Isabelle the induction has either to be done by the user or added into the automatic reasoning process by means of extending the automation that both proof assistants come with. There is also a more interactive way of proving in ACL2 but compared to the capabilities of Isabelle and Coq it lacks flexibility and expressive power.

One notable point is readability. Arguably ACL2 is the least readable of the three due to its use of the lisp syntax. Even simple theorems become quickly unreadable and one has to put a lot of effort into deconstructing larger lisp terms. Coq and Isabelle in that regard will feel a lot more natural to most users since they both use more common ML/Haskell-like syntax.

4 Concrete ACL2

The following chapter is build upon information found in [12] [4] [6].

4.1 Simple interaction with ACL2

The primary way of interacting with ACL2 is via the command line. Some examples of simple interaction with the ACL2 command line are given in listing 1. Every named input like the definition of a constant will be saved and can be used from there on out (listing 2).

```
ACL2(r) !>(+ 1 2)
3
ACL2(r) !>(and t nil t)
NIL
ACL2(r) !>(equal (list 1 2 3 4) (list 1 2 3 4))
T
```

Listing 1: simple interaction with the ACL2 command line/emacs

```
ACL2(r) !>(defconst *tmp* 6)

Summary
Form: ( DEFCONST *TMP* ...)
Rules: NIL
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
*TMP*
ACL2(r) !>(- 3 *tmp*)
-3
ACL2(r) !>
```

Listing 2: simple interaction with the ACL2 command line/emacs (2)

4.2 A simple example

This example is supposed to introduce the basics of ACL2 in a clear and simple way. First the function “app” will be introduced which appends two lists together. Then some simple theorems will be proven using the definition of “app”.

4.2.1 Function definition

```
(defun app (xs ys)
  (cond
    ((consp xs) (cons (car xs) (app (cdr xs) ys)))
    (t ys)
  )
)
```

Listing 3: definition of append

The function “app” (listing 3) is defined by using the keyword “defun” followed by the name of the function, a list of parameters and the function body. The two parameters “xs” and “ys” are the two lists to be concatenated where “xs” is the list in the front of the newly generated list. In the body the function checks whether the first list is empty and if so returns the second list. Otherwise it will deconstruct the first list “xs” (using

car which equals the head and cdr which equals the tail of a list) and adding the first element of “xs” to the result of the recursive call of app.

Upon sending this definition to the ACL2 command line the output in listing 4 is generated.

The console output shows that ACL2 proves the function is terminating.

```
ACL2(r) !>(defun app (xs ys)
  (cond
    ((consp xs) (cons (car xs) (app (cdr xs) ys)))
    (t ys)
  )
)

The admission of APP is trivial, using the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT XS). We observe that the type of APP is described by the
theorem (OR (CONSP (APP XS YS)) (EQUAL (APP XS YS) YS)). We used primitive
type reasoning.

Summary
Form: ( DEFUN APP ...)
Rules: ( (:FAKE-RUNE-FOR-TYPE-SET NIL) )
Time: 0.01 seconds (prove: 0.00, print: 0.00, other: 0.01)
APP
```

Listing 4: submission of app

For this it uses the well-founded relation “ $0<$ ” which is the the less-than relation on the ordinal numbers (which are represented by the predicate O-p) up to epsilon-0.

After termination has been proven, ACL2 then proves the well-typedness of the submitted term and presents the user with a term that describes the type of it. In this particular case the type reads “app applied to xs ys (or any two lists) is either a cons-cell or ys.”

Finally the ACL2 output shows a summary including the used theorems to prove the termination and typing.

After the definition has been successfully submitted ACL2 will silently generate several theorems including the termination, induction and typing theorem.

The definition of “app” can now be used in the shell interactively (listing 5).

```
ACL2(r) !>(app (list 1 2 3) (list 4 5 6))
(1 2 3 4 5 6)
```

Listing 5: app used in the ACL2 shell

4.2.2 Proofs using app

The first proof (listing 6) is a simple reduction theorem that states that appending the two lists “(list 1 2 3)” and “(list 4 5 6)” result in the list “(list 1 2 3 4 5 6)”. The summary tells the user information on how the proof was carried out. In this particular example the evaluation rules for

```

ACL2(r) !>(thm (equal (app (list 1 2 3) (list 4 5 6)) (list 1 2 3 4 5 6)))
Q.E.D.
Summary
Form: ( THM ...)
Rules: (:EXECUTABLE-COUNTERPART APP)
      (:EXECUTABLE-COUNTERPART EQUAL))
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
Prover steps counted: 5
Proof succeeded.

```

Listing 6: proof summary

“app” and “equal” were used to reduce their respective terms when applied to constants. The summary also tells the user that the proof succeeded. As an example of a failing proof, when the statement of the previous proof is modified such that it states that the two lists “(list 1 2 3)” and “(list 4 5 6)” result in the list “(list 1 2 3 4)” than the prover would print a similar summary with the indication that the proof has failed (listing 7). This time the trace is longer. Usually the prover would now print additional

```

ACL2(r) !>(thm (equal (app (list 1 2 3) (list 4 5 6)) (list 1 2 3 4)))
Goal'
([ A key checkpoint:
Goal'
NIL
A goal of NIL, Goal', has been generated! Obviously, the proof attempt
has failed.
])
Summary
Form: ( THM ...)
Rules: (:EXECUTABLE-COUNTERPART APP)
      (:EXECUTABLE-COUNTERPART EQUAL))
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
Prover steps counted: 7
---
The key checkpoint goal, below, may help you to debug this failure.
See :DOC failure and see :DOC set-checkpoint-summary-limit.
---
*** Key checkpoint at the top level
    before generating a goal of NIL (see :DOC nil-goal): ***
Goal'
NIL
ACL2 Error in ( THM ...): See :DOC failure.
***** FAILED *****

```

Listing 7: failing proof

information in an attempt to help the user understand what went wrong but for this failed proof the “extra” information just results in more verbosity without more information (since it is a very simple theorem). What the potential additional information will be and means will become clear in the rest of this work. For now the only important information is that the proof has failed.

One of the most important proof techniques is “proof by induction”. The theorem in listing 8 shows a proof that uses induction.

The first thing to notice is, that “app” is now applied to variables instead

```
ACL2(r) !>(defthm appassoc (equal (app xs (app ys zs)) (app (app xs ys) zs))
:hints (("Goal" :induct
          (app xs ys))))

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

We have been told to use induction. One induction scheme is suggested
by the induction hint.

We will induct according to a scheme suggested by (APP XS YS). This
suggestion was produced using the :induction rule APP. If we let
(:P XS YS ZS) denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (CONSP XS)) (:P XS YS ZS))
      (IMPLIES (AND (CONSP XS) (:P (CDR XS) YS ZS))
                (:P XS YS ZS)))).

This induction is justified by the same argument used to admit APP.
When applied to the goal at hand the above induction scheme produces
two nontautological subgoals.
Subgoal *1/2
Subgoal *1/1

*1 is COMPLETED!
Thus key checkpoint Goal is COMPLETED!

Q.E.D.

Summary
Form: ( DEFTHM APPASSOC ...)
Rules: (:DEFINITION APP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION APP)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS))
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
Prover steps counted: 255
APPASSOC
```

Listing 8: proof of induction for associativity of app

of concrete data. Those variables don’t have to be introduced to ACL2 due to it being a quantifier-less logic. Every variable is implicitly all-quantified. The second thing to notice are the “hints” in this proof. Hints are the primary way of influencing the behavior of the ACL2-prover. Hints are assigned to a (sub-)goal and from there on out passed to potentially newly generated sub-goals. In this case the hints only consist of information on how to do induction in this theorem.

Induction hints work by supplying the prover a term that is a recursively defined function applied to variables occurring in the theorem. The prover will then take the induction theorem generated from the passed function (listing 9) and apply the passed variables accordingly.

Back to the concrete problem at hands, the prover takes the induction hint

```
(AND (IMPLIES (NOT (CONSP XS)) (:P XS YS))
      (IMPLIES (AND (CONSP XS) (:P (CDR XS) YS))
                (:P XS YS)))
```

Listing 9: induction theorem generated by app

and generates two sub-goals (named “Subgoal *1/2” and “Subgoal *1/1”)

which are proven without any complications and thus the whole theorem is proven.

The appassoc theorem can now be used in other theorems (listing 10).

As can be seen in the summary under rules, ACL2 automatically uses the

```
ACL2(r) !>(thm (equal (app xs (app ys (app zs ls))) (app (app (app xs ys) zs) ls)))
Q.E.D.
Summary
Form: (THM ...)
Rules: (:REWRITE APPASSOC)
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
Prover steps counted: 30
Proof succeeded.
```

Listing 10: using appassoc in another theorem

appassoc theorem.

The next proof in listing 11 shows how typing (or the lack of) can influence a proof. The theorem itself states that appending an empty list (nil) to “xs” equals “xs”.

As the trace in listing 11 shows, this proof has failed. The problem is that

```
ACL2(r) !>(defthm appemp (equal (app xs nil) xs))
*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.
Perhaps we can prove *1 by induction. One induction scheme is suggested
by this conjecture.
We will induct according to a scheme suggested by (APP XS NIL). This
suggestion was produced using the :induction rule APP. If we let (:P XS)
denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (CONSP XS)) (:P XS))
      (IMPLIES (AND (CONSP XS) (:P (CDR XS)))
                (:P XS))).
This induction is justified by the same argument used to admit APP.
When applied to the goal at hand the above induction scheme produces
two nontautological subgoals.
([ A key checkpoint while proving *1 (descended from Goal):
Subgoal *1/2'
(IMPLIES (NOT (CONSP XS)) (NOT XS))
A goal of NIL, Subgoal *1/2'', has been generated! Obviously, the
proof attempt has failed.
...)
```

Listing 11: failed proof attempt

the type of “xs” is not fixed (so it actually is not necessarily a list). If “xs” is not a list the “consp xs” condition is also false. Hence “app xs nil” will evaluate to nil and not “xs”. To make this proof work one has to fix the type of “xs” by adding appropriate hypothesis as shown in listing 12.

An alternative would be to rewrite app like in listing 13. This works because now the type checking is part of the function body and if “xs” is not a list, then the function just returns xs.

```

ACL2(r) !>(defthm appemp (implies (true-listp xs) (equal (app xs nil) xs)))

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

Perhaps we can prove *1 by induction. Two induction schemes are suggested
by this conjecture. Subsumption reduces that number to one.

We will induct according to a scheme suggested by (APP XS NIL). This
suggestion was produced using the :induction rules APP and TRUE-LISTP.
If we let (:P XS) denote *1 above then the induction scheme we'll use
is
(AND (IMPLIES (NOT (CONSP XS)) (:P XS))
      (IMPLIES (AND (CONSP XS) (:P (CDR XS)))
                (:P XS))).
This induction is justified by the same argument used to admit APP.
When applied to the goal at hand the above induction scheme produces
three nontautological subgoals.

*1 is COMPLETED!
Thus key checkpoint Goal is COMPLETED!

Q.E.D.

```

Listing 12: fixing the type for xs

```

(defun app (xs ys)
  (cond
    ((not (true-listp xs)) xs)
    ((consp xs) (cons (car xs) (app (cdr xs) ys)))
    (t ys)
  )
)

```

Listing 13: alternative definition of app

4.3 A complex example

This second example is supposed to give a more in depth view of the mechanisms of ACL2 and present a more challenging scenario.

4.3.1 Function definitions

For this example the function “rev” will be defined which reverses a list (listing 14).

Submitting this to ACL2 functions will work as expected (listing 15). A

```
(defun rev (xs) (declare (xargs :guard (true-listp xs)))
  (cond
    ((consp xs) (app (rev (cdr xs)) (list (car xs))))
    (t nil)
  )
)
```

Listing 14: definition of rev

```
ACL2(r) !>(defun rev (xs) (declare (xargs :guard (true-listp xs)))
  (cond
    ((consp xs) (app (rev (cdr xs)) (list (car xs))))
    (t nil)
  )
)

The admission of REV is trivial, using the relation 0< (which is known
to be well-founded on the domain recognized by 0-P) and the measure
(ACL2-COUNT XS). We observe that the type of REV is described by the
theorem (OR (CONSP (REV XS)) (EQUAL (REV XS) NIL)). We used primitive
type reasoning and the :type-prescription rule APP.

Computing the guard conjecture for REV...

The guard conjecture for REV is trivial to prove, given primitive type
reasoning. REV is compliant with Common Lisp.

Summary
Form: (DEFUN REV ...)
Rules: (:FAKE-RUNE-FOR-TYPE-SET NIL)
(:TYPE-PRESCRIPTION APP)
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
REV
```

Listing 15: admission of rev

declare statement like in “rev” is used to introduce restrictions to a function. In the case of “rev” it is used to restrict the type of the parameter the function takes to be a list by using the “guard” statement. It is important to note that proofs will ignore declare statements. These statements are just useful for evaluation on the command line and for making the source code more self explanatory. To let the prover take information from declare statements into account, one would have to modify the actual body of the function. In this case “rev” needed to be modified like in listing 16.

```
(defun rev (xs) (declare (xargs :guard (true-listp xs)))
  (cond
    ((and (consp xs) (true-listp xs)) (app (rev (cdr xs)) (list (car xs))))
    (t nil)
  )
)
```

Listing 16: modified rev

4.3.2 Proofs using rev

The goal is to prove the statement in listing 17 which reads “the reverse of the reverse of a list xs is xs ”. For simplicity’s sake the typing is moved to the theorem itself hence the hypothesis “true-listp xs ” which ensures that xs is a list.

Submitting this theorem to ACL2 results in the trace in listing 18. ACL2

```
(thm (implies (true-listp xs)(equal (rev (rev xs)) xs)))
```

Listing 17: $\text{rev}(\text{rev } xs) = xs$

```
ACL2(r) !>(thm (implies (true-listp xs)(equal (rev (rev xs)) xs)))
*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

Perhaps we can prove *1 by induction. Two induction schemes are suggested
by this conjecture. These merge into one derived induction scheme.

We will induct according to a scheme suggested by (REV XS). This suggestion
was produced using the :induction rules REV and TRUE-LISTP. If we
let (:P XS) denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (AND (CONSP XS) (TRUE-LISTP XS)))
  (:P XS))
  (IMPLIES (AND (AND (CONSP XS) (TRUE-LISTP XS))
    (:P (CDR XS)))
    (:P XS))).

.

*** Key checkpoint at the top level: ***

Goal
(IMPLIES (TRUE-LISTP XS)
  (EQUAL (REV (REV XS)) XS))

*** Key checkpoint under a top-level induction
before generating a goal of NIL (see :DOC nil-goal): ***

Subgoal *1/2''
(IMPLIES (AND (CONSP XS)
  (TRUE-LISTP (CDR XS))
  (EQUAL (REV (REV (CDR XS))) (CDR XS)))
  (EQUAL (REV (APP (REV (CDR XS)) (LIST (CAR XS))))
    XS))

ACL2 Error in (THM ...): See :DOC failure.

***** FAILED *****
```

Listing 18: failed proof attempt for $\text{rev}(\text{rev } xs) = xs$

will try hard to prove this theorem but will eventually fail. The key-checkpoint “Subgoal *1/2”” reveals that ACL2 doesn’t know how “rev”

distributes over “app”. This leads to a new theorem in listing 19 which has to be proven first to make our initial proof work. The induction hint for the theorem is there to ensure that ACL2 will do induction over the appropriate variable. Otherwise ACL2 might choose *ys*.

This proof also fails (listings 19 and 20). The key-checkpoint “Subgoal

```
ACL2(r) !>(defthm revapp (implies (and (true-listp xs) (true-listp ys)) (equal (rev (app xs ys)) (app (rev ys) (rev xs)))) :hints (("Goal" :induct (app xs ys) )))

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

We have been told to use induction. One induction scheme is suggested
by the induction hint.

We will induct according to a scheme suggested by (APP XS YS). This
suggestion was produced using the :induction rule APP. If we let (:P XS YS)
denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (CONSP XS)) (:P XS YS))
      (IMPLIES (AND (CONSP XS) (:P (CDR XS) YS))
                (:P XS YS)))
...

Summary
Form: ( DEFTHM RESAPP ...)
Rules: (:DEFINITION APP)
       (:DEFINITION NOT)
       (:DEFINITION REV)
       (:DEFINITION TRUE-LISTP)
       (:ELIM CAR-CDR-ELIM)
       (:EXECUTABLE-COUNTERPART CONSP)
       (:EXECUTABLE-COUNTERPART REV)
       (:FAKE-RUNE-FOR-TYPE-SET NIL)
       (:INDUCTION APP)
       (:INDUCTION REV)
       (:INDUCTION TRUE-LISTP)
       (:REWRITE CAR-CONS)
       (:REWRITE CDR-CONS)
       (:TYPE-PRESCRIPTION APP))
Splitter rules (see :DOC splitter):
  if-intro: (:DEFINITION REV)
Time: 0.02 seconds (prove: 0.02, print: 0.00, other: 0.00)
Prover steps counted: 5583

---
The key checkpoint goals, below, may help you to debug this failure.
See :DOC failure and see :DOC set-checkpoint-summary-limit.
---
```

Listing 19: $\text{rev} (\text{app } xs \text{ } ys) = \text{app} (\text{rev } ys) (\text{rev } xs)$ (1)

*1/2”” (listing 20) might suggest a theorem proven earlier in listing 11. In the used rules section this rule is not listed which means that it has not been used at all. The reason is that *app* requires that the reverse of a list is also a list but ACL2 doesn’t know that yet. This leads to the theorem in listing 21.

With that, the theorem “revapp” can be proven successfully (listing 22).

The summary of the successful proof now shows that the rule “*app*” has been used. Now the initial theorem can be proven as seen in listing 23.

```

*** Key checkpoint at the top level: ***

Goal
(IMPLIES (AND (TRUE-LISTP XS) (TRUE-LISTP YS))
  (EQUAL (REV (APP XS YS))
    (APP (REV YS) (REV XS))))

*** Key checkpoint under a top-level induction
    before generating a goal of NIL (see :DOC nil-goal): ***

Subgoal *1/2''
(IMPLIES (TRUE-LISTP YS)
  (EQUAL (REV YS) (APP (REV YS) NIL)))

ACL2 Error in ( DEFTHM RESAPP ...): See :DOC failure.

***** FAILED *****

```

Listing 20: $\text{rev}(\text{app } xs \text{ } ys) = \text{app}(\text{rev } ys) (\text{rev } xs)$ (2)

```

ACL2(r) !>(defthm listrev (implies (true-listp xs) (true-listp (rev xs))))

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

...

*1.1 and *1 are COMPLETED!
Thus key checkpoints Subgoal *1/2'' and Goal are COMPLETED!

Q.E.D.

...

```

Listing 21: a proof that the reverse of a list is also a list

```

ACL2(r) !>(defthm revapp (implies (and (true-listp xs) (true-listp ys))
  (equal (rev (app xs ys)) (app (rev ys) (rev xs)))) :hints (("Goal" :induct (app xs ys) )))

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

...

Q.E.D.

...

Summary
Form: ( DEFTHM RESAPP ...)
Rules: ((:DEFINITION APP)
  (:DEFINITION NOT)
  (:DEFINITION REV)
  (:DEFINITION TRUE-LISTP)
  (:ELIM CAR-CDR-ELIM)
  (:EXECUTABLE-COUNTERPART CONSP)
  (:EXECUTABLE-COUNTERPART REV)
  (:FAKE-RUNE-FOR-TYPE-SET NIL)
  (:INDUCTION APP)
  (:INDUCTION REV)
  (:INDUCTION TRUE-LISTP)
  (:REWRITE APPEMP)
  (:REWRITE CAR-CONS)
  (:REWRITE CDR-CONS)
  (:REWRITE LISTREV)
  (:TYPE-PRESCRIPTION APP))
Splitter rules (see :DOC splitter):
  if-intro: ((:DEFINITION REV))
Time: 0.02 seconds (prove: 0.02, print: 0.00, other: 0.00)
Prover steps counted: 5000
REVAPP

```

Listing 22: successful proof of revapp

```

ACL2(r) !>(thm (implies (true-listp xs)(equal (rev (rev xs)) xs)))

*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

Perhaps we can prove *1 by induction. Two induction schemes are suggested
by this conjecture. These merge into one derived induction scheme.

We will induct according to a scheme suggested by (REV XS). This suggestion
was produced using the :induction rules REV and TRUE-LISTP. If we
let (:P XS) denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (AND (CONSP XS) (TRUE-LISTP XS)))
  (:P XS))
  (IMPLIES (AND (AND (CONSP XS) (TRUE-LISTP XS))
    (:P (CDR XS)))
    (:P XS))))).
This induction is justified by the same argument used to admit REV.
When applied to the goal at hand the above induction scheme produces
three nontautological subgoals.

*1 is COMPLETED!
Thus key checkpoint Goal is COMPLETED!

Q.E.D.

Summary
Form: (THM ...)
Rules: (:DEFINITION APP)
      (:DEFINITION NOT)
      (:DEFINITION REV)
      (:DEFINITION TRUE-LISTP)
      (:EXECUTABLE-COUNTERPART CONSP)
      (:EXECUTABLE-COUNTERPART EQUAL)
      (:EXECUTABLE-COUNTERPART REV)
      (:FAKE-RUNE-FOR-TYPE-SET NIL)
      (:INDUCTION REV)
      (:INDUCTION TRUE-LISTP)
      (:REWRITE CAR-CONS)
      (:REWRITE CDR-CONS)
      (:REWRITE CONS-CAR-CDR)
      (:REWRITE LISTREV)
      (:REWRITE REVAPP)
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
Prover steps counted: 469

Proof succeeded.

```

Listing 23: successful proof of $\text{rev} (\text{rev } xs) = xs$

5 Evaluation

ACL2 as a programming language is easy to pick up, since the language has a simple structure and simple typing system. This simplicity comes at a cost, though. Since it is achieved partly by using the Lisp syntax, defining functions and theorems becomes difficult to read very quickly. With regards to the typing system: The concept of types as predicates is easy to understand but hard to master. A lot of difficult to understand corner cases arise due to the way ACL2 handles typing. One different problem that arises from the way types work in ACL2 is that there is no real pattern-matching. This makes ACL2 in some situations rather clumsy.

Not having support for higher order functions and theorems also makes things quite often hard to formalize or proof.

Automatic proving is ACL2's strong suit but has a steep learning curve. Unlike virtually any other system, the automation in ACL2 is made so that a user can trace the steps the automatic prover takes in a proof attempt. The traces that ACL2 presents are with some training fairly understandable. Yet it often can be too much and every now and then the system behaves in a way that is incomprehensible to the user even when presented with a trace.

Instructing the system on how to behave or do things in a certain way requires a lot of thought, compared to a step-by-step proof construction like in Coq.

A problem that arises from primarily using automation for proving is that the system is not optimal for proof exploration or learning how to proof theorems. Since ACL2 always presents the user with a complete proof attempt one has to work through the trace by oneself instead of interactively work through the proof with the help of the system.

6 Conclusion/Future work

This work was just a short overview of the capabilities of ACL2. A lot of details were glossed over which make proving in ACL2 hard. The examples were also chosen so that things are understandable and don't get out of hand to quickly. The overview is far from being a "real world"-introduction to ACL2. More advanced and practical topics like hardware verification are completely missing.

Since proof assistants are often used to formalize mathematics it would be interesting how one would do such a thing in ACL2 and how this differs from using more classical provers such as Coq or Isabelle.

References

- [1] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.
- [2] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.
- [3] Toby Murray Daniel Matichuk, Makarius Wenzel. The eisbach user manual, 2018.
- [4] The ACL2 development team. ACL2 manual. In http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2_--_TOP. Dept. of Computer Sciences, University of Texas at Austin, 2020.
- [5] M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. In <http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz>. Dept. of Computer Sciences, University of Texas at Austin, 1997.
- [6] M. Kaufmann and J S. Moore. How to proof theorems formally, 2005.
- [7] M. Kaufmann and J S. Moore. Industrial proofs with acl2, 2005.
- [8] Tobias Nipkow. What's in main, 2017.
- [9] Tobias Nipkow. Programming and proving in Isabelle/HOL, 2020.
- [10] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.
- [11] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5. <http://www.cis.upenn.edu/~bcpierce/sf>.
- [12] Pucella. Using the acl2 theorem prover, 2009.
- [13] The Coq Development Team. The coq reference manual. In <https://coq.inria.fr/refman/>. Inria, 2020.
- [14] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag, Berlin, Heidelberg, 2006.