IA159 Formal Verification Methods

Theorem Prover ACL2

Jan Strejček

Faculty of Informatics
Masaryk University

Theorem provers

Theorem provers are software tools, which

- assist human experts in construction of formal proofs
- can be used in software and hardware verification
- work only with statements (and corresponding axioms and inference rules) in a suitable formal notation

Do they work automatically?

- only simple theorems can be proven fully automatically
- nearly all proofs result from interaction of a tool and a user
- success depends primarily on user's skill

ACL2

"A Computational Logic for Applicative Common Lisp"

ACL2 is

1. a functional programming language based on Common Lisp
2. a first-order, quantifier-free mathematical logic
3. a mechanical theorem prover

History of ACL2

- 1971: Robert S. Boyer and J Strother Moore created Nqthm - the first theorem prover for Lisp
- 1989: Boyer and Moore started to work on ACL2
- since 1993, ACL2 is systematically developed by Matt Kaufmann and J Strother Moore
- now in version 8.1 (February 2019)
- winner of VSTTE 2012 Software Verification Competition

ACL2 is available under a license based on BSD-3-Clause

`http://www.cs.utexas.edu/users/moore/acl2/`

ACL2 achievements

ACL2 has been used to verify that

- the functionality of FPU in AMD K5, Athlon, and Opteron (described on register-transfer level, RTL) follows the corresponding IEEE standard
- the microarchitectural model of a Motorola DSP processor implements a given microcode engine and that certain microcode in ROM implements certain DSP algorithms
- the microcode for the Rockwell Collins AAMP7 implements a given security policy concerning process separation
- the bytecode produced by the Sun compiler `javac` on certain simple Java classes has the claimed functionality
- a BDD package written in Lisp is sound and complete
- a Lisp program that checks proofs produced by the Ivy theorem prover is sound
- ...

Back to the ground

- the proof of correctness of the floating point division microcode in AMD K5 required approx. 1200 lemmas
- the verification has not been done directly on RTL of the FPU, but on its automatically translated Lisp model; correctness of the translation has been “verified” by 80 000 000 test computations
- the proof of correspondence between the Motorola DSP microarchitecture and its microcode engine involved formulas of 25 MB per formula; finding one subtle generalization took Moore many days
- …

The mechanics of using ACL2

Two buffers in Emacs

1. buffer with definitions and lemmas, typically concluding with the main theorem we want to prove
2. shell with ACL2

Typical working cycle

1. send the subsequent definition or theorem to ACL2
2. if it succeeds, go to 1
3. inspect the output of ACL2 and analyze the failure
4. if the command is faulty (e.g. with a syntax error), fix it; if the command is a theorem ACL2 is unable to prove, then suggest, formulate, and prove additional lemmas and then try to prove the theorem again

Syntax of ACL2

Supported data objects

- numbers (integer, rational and complex)
- characters
- strings (`"Hello world!"`)
- symbols (`t`, `nil`, `'ok`, `'quick-sort`,...)
- ordered pairs

Lists

- in fact nested pairs: $\langle 1, \langle 2, \langle 3, \texttt{nil} \rangle\rangle\rangle$ or $\langle 1, \langle 2, 3 \rangle\rangle$
- written in list notation: `'(1 2 3)` or `'(1 2 . 3)`

Syntax of ACL2

Some primitive (built-in) functions

`(cons` *x y*`)`	constructs the ordered pair $\langle x, y \rangle$
`(car` *x*`)`	left component of *x*, if *x* is a pair; `nil` otherwise
`(cdr` *x*`)`	right component of *x*, if *x* is a pair; `nil` otherwise
`(consp` *x*`)`	`t` if *x* is a pair; `nil` otherwise
`(if` *x y z*`)`	*z* if *x* is `nil`; *y* otherwise
`(equal` *x y*`)`	`t` if *x* is *y*; `nil` otherwise

Meaning of the single quote mark $\boxed{'}$

- `'(car x)` evaluates to the list $\langle \texttt{car}, \langle \texttt{x}, \texttt{nil} \rangle\rangle$
- `(car x)` application of the function `car` to `x`

Syntax of ACL2

Function definition

- `(defun` f `(`a_1 a_2 `...` a_n`)` β`)`
  creates the function f with arguments $a_1, a_2, \ldots, a_n$
  and body β

(Built-in) Lisp definitions of standard logic connectives

- `(defun not (p) (if p nil t))`
- `(defun and (p q) (if p q nil))`
- `(defun or (p q) (if p p q))`
- `(defun implies (p q) (if p (if q t nil) t))`
- ```
  (defun iff (p q) (and (implies p q)
                        (implies q p)))
  ```
  ```

Examples of recursive function definitions

- `dup` - duplicates each element in a list

```
(defun dup (x)
  (if (consp x)
      (cons (car x)
            (cons (car x)
                  (dup (cdr x))))
      nil))
```

- `app` - concatenates two lists

```
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
      y))
```

Axioms in ACL2

Some primitive (built-in) axioms

1. $\texttt{t} \neq \texttt{nil}$
2. $\texttt{x} \neq \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{y}$
3. $\texttt{x} = \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{z}$
4. $\texttt{(equal x y)} = \texttt{nil} \lor \texttt{(equal x y)} = \texttt{t}$
5. $\texttt{x} = \texttt{y} \leftrightarrow \texttt{(equal x y)} = \texttt{t}$
6. $\texttt{(consp x)} = \texttt{nil} \lor \texttt{(consp x)} = \texttt{t}$
7. $\texttt{(consp (cons x y))} = \texttt{t}$
8. $\texttt{(consp nil)} = \texttt{(consp t)} = \texttt{(consp 'ok)} = \texttt{(consp 0)} = \ldots = \texttt{nil}$
9. $\texttt{(car (cons x y))} = \texttt{x}$
10. $\texttt{(cdr (cons x y))} = \texttt{y}$
11. $\texttt{(consp x)} = \texttt{t} \rightarrow \texttt{(cons (car x) (cdr x))} = \texttt{x}$

Proofs in ACL2 (a sketch)

ACL2 contains

- ordinals up to $\omega^{\omega^{\omega^{\cdots}}}$
- a well-founded relation `o<` on such ordinals
- axioms defining the size of ACL2 objects (measured with the function `acl2-count`)

and particularly

- definition principle
- induction principle
- simplification based on
  - rewrite rules
  - linear arithmetic rules (inequality chaining)
  - ... (approx. 12 kinds of rules in total)

Definition principle

- when a recursive function definition is submitted, ACL2 must prove that there is a well-founded measure such that arguments of recursive calls are decreasing with respect to this measure
- existence of such a measure ensures that the evaluation of the function terminates after a finite number of steps.
- the definition is admitted by ACL2 (as a new axiom) only if the existence of such a measure is proven; a user can assist with the proof

Induction principle (a sketch)

Structural induction on lists and binary trees

If we want to prove $\varphi(x, y)$, it is sufficient to prove

1. base case
   $\varphi(x, y)$ holds in the case that x is an empty tree
2. induction step
   if $x = (l, r)$ and $\varphi(l, y)$ and $\varphi(r, y)$, then $\varphi(x, y)$

Induction principle (a sketch)

Induction on binary trees in ACL2

If we want to prove `(`φ` x y)`, it is sufficient to prove

1. base case

```
(implies (not (consp x)) (φ x y))
```

2. induction step

```
(implies (and (consp x)
              (φ (car x) y)    ;induction hypothesis 1
              (φ (cdr x) y))   ;induction hypothesis 2
         (φ x y))              ;induction conclusion
```

- induction hypothesis can be any `(`φ δ `y)` such that we can prove `(implies (consp x)`
  `(o< (acl2-count` δ`) (acl2-count x))`
- axioms imply that `(car x)`, `(cdr x)` are smaller than `x`

A proof

```
(defun treecopy (x)
  (if (consp x)
      (cons (treecopy (car x))
            (treecopy (cdr x)))
     x))
```

Theorem: `(equal (treecopy x) x).`

Proof: Name the formula above *1.

Perhaps we can prove *1 by induction. One induction scheme is suggested by this conjecture.

A proof

We will induct according to a scheme suggested by `(treecopy x)`. This suggestion was produced using the induction rule `treecopy`. If we let `(φ x)` denote *1 above then the induction scheme we'll use is

```
(and (implies (not (consp x)) (φ x))
     (implies (and (consp x)
                   (φ (car x))
                   (φ (cdr x)))
              (φ x))).
```

A proof

This induction is justified by the same argument used to admit `treecopy`. When applied to the goal at hand the above induction scheme produces two nontautological subgoals.

Subgoal *1/2

```
(implies (not (consp x))
         (equal (treecopy x) x)).
```

But simplification reduces this to `t`, using the definition `treecopy` and primitive type reasoning.

A proof

Subgoal *1/1

```
(implies (and (consp x)
              (equal (treecopy (car x)) (car x))
              (equal (treecopy (cdr x)) (cdr x)))
         (equal (treecopy x) x)).
```

But simplification reduces this to `t`, using the definition `treecopy`, primitive type reasoning and the rewrite rule `cons-car-cdr`.

That completes the proof of *1.

Q.E.D. □

Simplification of Subgoal *1/1

```
(implies (and (consp x)                                   ;hypothesis 1
              (equal (treecopy (car x)) (car x))   ;hypothesis 2
              (equal (treecopy (cdr x)) (cdr x))) ;hypothesis 3
         (equal (treecopy x) x)).
```

```
(treecopy x) = (if (consp x)                          ;treecopy definition
                    (cons (treecopy (car x))
                          (treecopy (cdr x)))
                   x)
             = (if t                                  ;hypothesis 1
                    (cons (treecopy (car x))
                          (treecopy (cdr x)))
                   x)
             = (cons (treecopy (car x))               ;axioms 1 and 2
                     (treecopy (cdr x)))
             = (cons (car x)                          ;hypothesis 2
                     (treecopy (cdr x)))
             = (cons (car x)                          ;hypothesis 3
                     (cdr x))
             = x                                      ;axiom 11 and
                                                      ;hypothesis 1
```

Proofs in ACL2: induction

- ACL2 uses heuristics to choose a suitable induction scheme
- induction scheme is based on recursively defined function occurring in the theorem
- the resulting scheme can be a combination of two or more recursive schemes used in the theorem
- the user can specify an induction scheme with a hint
- choosing the right induction is crucial to a successful proof
- even more important is to choose the right theorem to prove by induction - the theorem has to be general enough in order to provide a sufficiently strong induction hypothesis

Proofs in ACL2: simplification via rewriting

- simplification means the reduction of the formula to some preferred form by the use of rewrite rules
- rules are derived from axioms, definitions, and previously proved theorems
- a definition generates the rule rewriting function calls by the instantiated body of the function
- a formula of the form

  $$\texttt{(implies (and } \mathit{hyp}_1 \ldots \mathit{hyp}_n\texttt{) (equal } l\ r\texttt{))}$$

  generates the rule replacing instances of l by the corresponding instance of r, provided the corresponding instances of $\mathit{hyp}_1, \ldots, \mathit{hyp}_n$ rewrite to `t`
- equivalent formulae (like `(equal` l r`)` and `(equal` r l`)`); may give rise to radically different rules
- some rule combinations can lead to cyclic rewriting

Proofs in ACL2: simplification via inequality chaining

- there is a large set of rewrite rules allowing to put arithmetic expressions into a preferred form
- there are books (i.e. collections of such rules) for elementary algebraic properties of numbers, modulo arithmetic, floating point arithmetic, ...

- ACL2 also maintains a graph of terms involved in the current formula, where edges correspond to inequalities
- edges are added by a decision procedure for linear arithmetics
- when submitting a theorem, we can specify what kind of rule should be generated when the theorem is proved (it is a rewrite rule by default)

Proofs in ACL2: simplification via inequality chaining

Example

- consider a theorem concluding with `(<= 0 (* x x))`
- if it is used to generate a rewrite rule, the rule replaces certain instances of `(<= 0 (* x x))` by `t`
- this rule cannot be used to prove

```
(implies (and (< 0 a) (rationalp b))
         (< 0 (+ a (* b b))))
```

- if we say that an arithmetic rule should be gerenated from the theorem, the the rule can be used to add some edges to the graph of inequalities

There are many other kinds of simplification.

ACL2 working cycle

- when ACL2 receives a syntactically correct theorem, it
  1. simplifies the theorem
  2. uses an induction
  3. go to 1
- it exits the cycle if
  - the simplification results in `t`, or
  - there is no suggested induction scheme
- if the theorem is proved, a corresponding rule is derived
- ...and everything is vividly commented

Demonstration

Coming next week

Reachability in pushdown system

- How can I denote an infinite-state system?
- Can I verify an infinite-state system?
- What are pushdown processes good for?