Introduction to ACL2

CS 680 Formal Methods for Computer Verification

Jeremy Johnson
Drexel University
ACL2 is a programming language, logic, and theorem prover/checker based on Common Lisp.

ACL2 is a powerful system for integrated modeling, simulation, and inductive reasoning. Under expert control, it has been used to verify some of the most complex theorems to have undergone mechanical verification.
ACL2s (acl2s.ccs.neu.edu)

- Eclipse plugin (sedan version)
  - Pure functional subset
  - Ensure valid input
  - Different operational modes
  - Termination analysis
  - Random testing and bug generation
  - Install and test and read (ACL2s Programming Language)
Read-Eval-Print-Loop (REPL)

- ACL2s reads inputs, evaluates them and prints the result

ACL2S BB !>VALUE (* 2 3)
6
ACL2S BB !>
A Pure Functional Language

- $x_1 = y_1, \ldots, x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$

- No side-effects, no assignments, no state, no loops
- Use recursion instead of iteration
- Still Turing complete
- Makes reasoning about programs easier
C++ Function with Side-Effects

```cpp
#include <iostream>
using namespace std;

int cc()
{
    static int x = 0;
    return ++x;
}

int main()
{
    cout << "cc() = " << cc() << endl;
    cout << "cc() = " << cc() << endl;
    cout << "cc() = " << cc() << endl;
    % g++ count.c
    % ./a.out
    cc() = 1
    cc() = 2
    cc() = 3
}
```
ACL2 Syntax and Semantics

- Atoms (symbols, booleans, rationals, strings)
  - predicates
- Lists ((1 2) 3)
  - nil, cons, first and rest
- Functions and function application
  - (* 2 (+ 1 2))
- If expressions
  - (if test then else)
ACL2 Atoms

- Rationals: For example, 11, -7, 3/2, -14/15
- Symbols: For example, x, var, lst, t, nil
- Booleans: There are two Booleans, t, denoting true and nil, denoting false
- Strings: For example, “hello”, “good bye”
Function Application

- (* 2 3)
  - 6
- (* 2 (+ 1 2))
  - 6
- (numerator 2/3)
  - 2
- (f x₁ … xₙ) [applicative order]
if expressions

- if : Boolean × All × All → All

- (if test then else)

- (if test then else) = then, when test = t
- (if test then else) = else, when test = nil
Example if expressions

- (if t nil t)
- (if nil 3 4)
- (if (if t nil t) 1 2)
Equal

- $\text{equal} : \text{All} \times \text{All} \rightarrow \text{Boolean}$

- $(\text{equal } x \ y)$ is $t$ if $x = y$ and $\text{nil}$ otherwise.

- $(\text{equal } 3 \ \text{nil}) = \text{nil}$
- $(\text{equal } 0 \ 0) = t$
- $(\text{equal } (\text{if } t \ \text{nil} \ t) \ \text{nil}) = t$
Predicates

- All → Boolean
- booleanp
- symbolp
- integerp
- rationalp
Defining Functions

(defun booleanp (x)
  (if (equal x t)
      t
    (equal x nil)))
Input/Output Contracts

(defunc booleanp (x)
  :input-contract t
  :output-contract (booleanp (booleanp x))
  (if (equal x t)
    t
    (equal x nil))))
Input/Output Contracts

- $ic \Rightarrow oc$

- For booleanp (type checking)
  - $\forall x :: t \Rightarrow (booleanp (booleanp x))$
  - $\forall x :: (if t (booleanp (booleanp x)) t)$
  - $\forall x :: (booleanp (booleanp x))$
Contract Checking

- ACL2s will not admit a function unless it can prove that every function call in its body satisfies its contract (body contract checking) and can show that it satisfies its contract (contract checking)
Contract Violations

ACL2S BB !>VALUE (unary-/ 0)

ACL2 Error in ACL2::TOP-LEVEL: The guard for the function call (UNARY-/ X), which is (COMMON-LISP::AND (RATIONALP X) (COMMON-LISP::NOT (EQUAL X 0))), is violated by the arguments in the call (UNARY-/ 0).
Contract Checking Example

(defun foo (a)
  :input-contract (integerp a)
  :output-contract (booleanp (foo a))
(if (posp a)
  (foo (- a 1))
  (rest a))))
Boolean Functions

- And : Boolean × Boolean → Boolean
  
  (defunc and (a b)
    :input-contract (if (booleanp a) (booleanp b) nil)
    :output-contract (booleanp (and a b))
  (if a b nil))

- Or

- Not

- Implies

- Iff

- Xor
Numbers

- *, +, <, unary--, unary-/

(defunc unary-/ (a)
  :input-contract (and (rationalp a) (not (equal a 0)))
...
)

- Numerator, Denominator

- Exercise: Subtraction and Division
(posp)

(defun posp (a)
  :input-contract t
  :output-contract (booleanp (posp a))
  (if (integerp a)
      (< 0 a)
      nil))
Incorrect posp

(defunc posp (a)
  :input-contract t
  :output-contract (booleanp (posp a))
(and (integerp a)
  (< 0 a)))
Termination?

- ACL2 will only accept functions that it can prove terminate for all inputs
- Does the following always terminate?

;; Given integer n, return 0+1+2+...+n
(defun sum-n (n)
  :input-contract (integerp n)
  :output-contract (integerp (sum-n n))
  (if (equal n 0)
      0
      (+ n (sum-n (- n 1))))
Termination?

- Modify the input-contract so that sum-n does terminate for all inputs

;; Given integer n, return 0+1+2+...+n
(defun sum-n (n)
  :input-contract (integerp n)
  :output-contract (integerp (sum-n n))
  (if (equal n 0)
      0
      (+ n (sum-n (- n 1))))
Modify the input-contract so that sum-n does terminate for all inputs

;; Given integer n, return 0+1+2+...+n
(defun sum-n (n)
  :input-contract (natpp n)
  :output-contract (integerp (sum-n n))
  (if (equal n 0)
      0
      (+ n (sum-n (- n 1))))
natp

;; Test whether the input is a natural number (integer ≥ 0)
(defunc natp (a)
  :input-contract t
  :output-contract (booleanp (natp a))
  (if (integerp a)
    (or (< 0 a) (equal a 0))
    nil))
Lists: \( L = (x_1 \ldots x_n) \)

- A list is either empty (\( () \) [same as nil]) or made up of Cons cells
- consp : All \( \rightarrow \) Boolean
- cons : All \( \times \) List \( \rightarrow \) Cons
  - \((\text{cons} \ x \ L) = (x \ x_1 \ldots x_n)\)
- first : Cons \( \rightarrow \) All
  - \((\text{first} \ (\text{cons} \ x \ L)) = x\)
- rest : Cons \( \rightarrow \) List
  - \((\text{rest} \ (\text{cons} \ x \ L)) = L\)
List Examples

- (list x₁ ... xₙ) [macro – variable # of args]
- (cons x₁ (cons x₂ ... (cons xₙ nil)...) )
- (consp ()) = nil
- (cons 1 ()) = (1)
- (consp (cons 1 ())) = t
- (cons 1 (cons 2 (cons 3 ()))) = (1 2 3)
- (list 1 2 3) = (1 2 3)
- (cons (cons 1 ()) ()) = ((1))
Exercise

- How do you construct the list ((1) 2)?
Exercise

- How do you construct the list ((1) 2)?
  - (cons 1 ()) = (1)
  - (cons 2 ()) = (2)
- Want (cons x y) where x = (1) and y = (2)
  - (cons (cons 1 ()) (cons 2 ())) = ((1) 2)
- Test this (and try additional examples) in ACL2 REPL
More About cons

- A cons cell contains two fields
  - first  [also called car]
  - rest   [also called cdr]

- For a list the rest field must be a list
- Generally both fields of a cons $\in$ All
  - (cons 1 2) = (1 . 2)
  - Called a dotted pair
List Processing

- Recursive Definition
  - List : nil | cons All List
  - Process lists using these two cases and use recursion to recursively process lists in cons
- Use first and rest to access components of cons
Recursion and listp

(defun listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
(if (consp l)
  (listp (rest l))
  (equal l () )))

1. listp is a built-in function in the full ACL2 mode that only checks for consp. True-listp behaves as above.
Incorrect listp

- Nonterminating
- ACL will not accept functions it cannot prove terminate

(defun listp (a)
  :input-contract t
  :output-contract (booleanp (listp a))
(if (consp a)
  (listp a)
  (equal a nil)))
Version 2 of listp

(defun listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (if (equal l nil)
    t
    (if (consp l)
      (listp (rest l))
      nil))))
Version 3 of listp

(defun listp (l)
  :input-contract t
  :output-contract (booleanp (listp l))
  (cond
   ((equal l nil) t)
   ((consp l) (listp (rest l)))
   (nil)))
Length¹

(defun length (l)
 :input-contract (true-listp l)
 :output-contract (natp (length l))
 (if (equal l nil)
      0
      (                              )))

¹. Length is a built-in function in the full ACL2 mode
Length

(defun length (l)
 :input-contract (true-listp l)
 :output-contract (natp (length l))
 (if (equal l nil)
  0
  (+ 1 (length (rest l)))) )

- The recursive function can be “thought of” as a definition of length
Member

(defunc member (x l)
 :input-contract (true-listp l)
 :output-contract (booleanp (member x l))
 (cond
   ((equal l nil) nil)
   ( ... ))
 ( ... )))

1. Member is a built-in function in the full ACL2 mode
(defun member (x l)
  :input-contract (true-listp l)
  :output-contract (booleanp (member x l))
  (cond
   ((equal l nil) nil)
   ((equal x (first l)) t)
   ((member x (rest l))))
Append

(defun append (x y)
  :input-contract (and (true-listp x) (true-listp y))
  :output-contract (true-listp (append x y))
  ...
)

- (append '(1 2 3) '(4 5 6)) → (1 2 3 4 5 6)
- Recurse on the first input

1. Append is a built-in function in the full ACL2 mode
Append

(defun append (x y)
  :input-contract
  (if (equal x nil)
      y
      (cons (first x) (append (rest x) y))))
Reverse¹

(defun reverse (l)
  :input-contract (true-listp l)
  :output-contract (true-listp (reverse l))
  ...
  )

- (reverse '(1 2 3)) → (3 2 1)

1. Reverse is a built-in function in the full ACL2 mode
Reverse

(defun reverse (l)
  :input-contract (true-listp l)
  :output-contract (true-listp (reverse l))
  (if (equal l nil)
      nil
      (append (reverse (rest l)) (cons (first l) nil)))))
Numatoms

(defun atomp (x)
 :input-contract t
 :output-contract (booleanp (atomp x))
  (not (consp x)))

(defun numatoms (x)
 :input-contract t
 :output-contract (natp (numatoms x))
  (cond
   ((equal x nil) 0)
   ((atomp x) 1)
   ((+ (numatoms (first x)) (numatoms (rest x)))))))
Shallow vs Deep Recursion

- Length and Member only recurse on the rest field for lists that are conses
  - Such recursion is called shallow – it does not matter whether the lists contain atoms or lists
  - \( \text{length '((1 2) 3 4)) = 3 \)

- Numatoms recurses in both the first and rest fields
  - Such recursion is called deep – it completely traverses the list when elements are lists
  - \( \text{numatoms '((1 2) 3 4)) = 4 \)