Inductive Assertions in ACL2: Motivation and Ingredients

Sandip Ray

Department of Computer Sciences
University of Texas at Austin
sandip@cs.utexas.edu
http://www.cs.utexas.edu/users/sandip

Broad Goal

Facilitate the use of theorem proving to prove correctness of sequential programs running on machines modeled operationally in the logic.

Operational semantics:

- `(step s)` returns the state after executing one instruction from s.
- `(run s n)` returns the state after execution of n instructions.

```
(defun run (s n)
  (if (zp n)
      s
    (run (step s) (- n 1))))
```

Verification of Sequential Programs

Partial Correctness:

If the program is initiated from a machine state satisfying a given `precondition`, then if the program reaches a `halting` state, it satisfies the desired `postcondition`.

```
(defun halting (s) (equal s (step s)))

(defthm partial-correctness
  (implies (and (pre s)
                (halting (run s n)))
           (post (run s n))))
```

Verification of Sequential Programs

Total Correctness:

If the program is initiated from a machine state satisfying a given `precondition`, then the program reaches a `halting` state, and it satisfies the desired `postcondition`.

```
(defun-sk exists-halting-state
  (exists n (halting (run s n))))

(defthm termination
  (implies (pre s)
           (exists-halting-state s)))
```

Total correctness is partial correctness together with termination.

Total Correctness

Total Correctness: Alternative Formulation

```
(defthm total-correctness
  (implies (pre s)
           (and (halting (run s (clock s)))
                (post (run s (clock s))))))
```

This leads to the so-called “clock function proofs” in the Boyer-Moore community.

Halting Points vs. Exitpoints

Correctness:

If the program is initiated from a machine state satisfying a given `precondition`, then if the program reaches a `halting` state, it satisfies the desired `postcondition`.

More commonly, we talk about `exit` states, the predicate `exit` characterizes the exitpoints of subroutines or other program blocks.

In that case, we want to assert the postcondition on the first reachable `exit` state from a `pre` state.

Partial Correctness Using Exitpoints

```
(defthm partial-correctness
  (implies (and (pre s)
                (natp n)
                (exit (run s n)))
           (and (exit (run s (clock s)))
                (post (run s (clock s)))
                (natp (clock s))
                (<= (clock s) n))))
```

Thus `clock` takes us to the first `exit` state (if one is reachable), and the postcondition holds there.

Proving Partial Correctness: Traditional ACL2 Way

Define `inv` such that:

```
(defthm pre-implies-inv
  (implies (pre s) (inv s)))

(defthm inv-is-inductive
 (implies (and (inv s) (not (exit s)))
          (inv (step s))))

(defthm inv-is-sufficient
  (implies (and (inv s) (exit s))
           (post s)))
```

Predicate `inv` is often called an **inductive invariant**.

Proving Termination: Traditional ACL2 Way

Define a function `rank` such that:

```
(defthm rank-is-ordinal
;; Can assume (inv s) here but is usually not necessary
  (o-p (rank s)))

(defthm rank-decreases
  (implies (and (inv s)
                (not (exit s)))
           (o< (rank (step s))
               (rank s))))
```

Function `rank` is often called the **ranking function**.
Another (more common) approach is simply to define a `clock`.

What is complicated about these proofs?

A simple program:

```
1:    X:=0                    {T}
2:    Y:=10
3:    if (Y ≤ 0) goto 7
4:    X:=X+1
5:    Y:=Y-1
6:    goto 3
7:    HALT                    {(X=10)}
```

What is complicated about these proofs?

Correctness proof using Step Invariants

```
1:    X:=0                  {T}
2:    Y:=10
3:    if (Y ≤ 0) goto 7
4:    X:=X+1
5:    Y:=Y-1
6:    goto 3
7:    HALT                  {(X=10)}
```

The predicate `inv` needs to characterize every reachable state.

What is complicated about these proofs?

Partial correctness proof using Step Invariants

1:	X:=0	$\{T\}$
2:	Y:=10	$\{(X=0)\}$
3:	if (Y $\leq$ 0) goto 7	$\{(X+Y=10)\}$
4:	X:=X+1	$\{(Y > 0) \wedge (X+Y=10)\}$
5:	Y:=Y-1	$\{(Y > 0) \wedge (X+Y=11)\}$
6:	goto 3	$\{(Y \geq 0) \wedge (X+Y=10)\}$
7:	HALT	$\{(X=10)\}$

The predicate `inv` needs to characterize every reachable state.

A Typical ACL2 Definition

```
(defun inv (s)
  (case (pc s)
   (0 ...)
   (1 ...)
   (2 ...)
    ....
   (7 ...)))
```

Too tedious!! Also often complicated to figure out what we should write at every program counter value.

The same is actually also true for the `rank`.

Towards more Automation: Assertional Reasoning

Annotate the program only at cutpoints.

```
1:    X:=0                 {T}
2:    Y:=10
3:    if (Y ≤ 0) goto 7    {(X+Y=10)}
4:    X:=X+1
5:    Y:=Y-1
6:    goto 3
7:    HALT                 {(X=10)}
```

Cutpoints are loop tests and program entry and exit points.

Assertional Reasoning

Annotate the program only at cutpoints.

```
1:   X:=0                  {T}
2:   Y:=10
3:   if (Y ≤ 0) goto 7     {(X+Y=10)}
4:   X:=X+1
5:   Y:=Y-1
6:   goto 3
7:   HALT                  {(X=10)}
```

A Verification Condition Generator (VCG) crawls over the annotated program to generate certain formulas.

{**T**} X:=0; Y:=10 {**(X+Y)=10**}

T $\Rightarrow$ **(0+10)=10**

Uses (implicitly) a bunch of axioms: $\{\Phi(b)\}$ x:= b $\{\Phi(x)\}$.

Assertional Reasoning

Annotate the program only at cutpoints.

```
1:   X:=0                 {T}
2:   Y:=10
3:   if (Y ≤ 0) goto 7    {(X+Y=10)}
4:   X:=X+1
5:   Y:=Y-1
6:   goto 3
7:   HALT                 {(X=10)}
```

A verification condition generator crawls over the annotated program to generate certain formulas.

$\{\mathbf{T}\}$ X:=0; Y:=10 $\{\mathbf{(X+Y)=10}\}$

$\mathbf{T} \Rightarrow \mathbf{(0+10)=10}$

- The verification conditions can be proven by a theorem prover.

Goal of the Project

- We will attach assertions (and ranking functions) **only** at cutpoints.
- We will prove partial (and total) correctness for operationally modeled programs.
- We will **not** implement or verify a VCG.

Key Observation

Moore (2003): Given assertions at cutpoints, we can generate an inductive invariant.

```
(inv s)
=
(if (cutpoint s)
    (assertion s)
  (inv (step s)))
```

Notice that the “definition” of `inv` is recursive, but **might not** terminate.

No problem!! The definition is tail-recursive and hence admissible in ACL2.

Key Ingredient: defpun

We can use `defpun` to write any tail-recursive “definitions” in ACL2.

Manolios and Moore (2003): Any tail-recursive definition is admissible (whether terminating or not).

Such definitions can even be executed, but we’ll not get into that.

Why are tail-recursive definitions ok?

Suppose you have arbitrary functions `(test x)`, `(base x)`, and `(recur x)`.

How do you define a function `f` such that the following is a theorem?

```
(equal (f x)
       (if (test x)
           (base x)
         (f (recur x))))
```

Why are tail-recursive definitions ok?

We can clearly define a “bounded version” of `f`:

```
(defun fn (x n)
  (if (or (zp n) (test x))
      (base x)
    (fn (recur x) (1- n))))
```

The function `fn` recurs until n becomes `0` or `test` becomes true.

Why are tail-recursive definitions ok?

We can clearly define a “bounded version” of `f`:

```
(defun fn (x n)
  (if (or (zp n) (test x))
      (base x)
    (fn (recur x) (1- n))))
```

The function `fn` recurs until n becomes `0` or `test` becomes true.

We now want to choose a large enough n.

What is large enough?

We don’t know how large is necessary, but we can provide a large n using quantification.

```
(defun recur-n (x n)
  (if (zp n) x
    (recur-n (recur x) (1- n))))

(defchoose choice (n) (x)
  (test (recur-n x n)))

(defthm choice-is-large-enough
  (implies (test (recur-n x n))
           (test (recur-n x (choice x)))))
```

Defining f

```
(defun f (x)
  (if (test (recur-n x (choice x)))
      (fn x (choice x))
    42 ;; or "J Moore" or any constant you like
   ))
```

The key theorem:

```
(defthm original-terminates-iff-recursive
  (implies (not (test x))
           (equal (test (recur-n (recur x)
                                 (choice (recur x))))
                  (test (recur-n x (choice x))))))
```

Thus the original recursion terminates iff the recursive call does.

Defpun

```
(defpun f (x)
  (if (test x)
      (base x)
    (f (recur x))))
```

Generates a function f which is constrained to satisfy the following axiom:

```
(equal (f x)
       (if (test x)
           (base x)
         (f (recur x))))
```

This is done by following the recipe of defining `f` above.

Moore's Invariant

```
(defpun inv (s)
  (if (cutpoint s)
      (assertion s)
    (inv (step s))))
```

Now what happens if you try to prove that it is an inductive invariant?

Recall that the theorem we want to prove is:

```
(defthm inv-is-inductive
 (implies (and (inv s) (not (exit s)))
          (inv (step s))))
```

Using Moore's Method

```
1:   X:=0                    {T}
2:   Y:=10
3:   if (Y ≤ 0) goto 7       {(X+Y=10)}
4:   X:=X+1
5:   Y:=Y-1
6:   goto 3
7:   HALT                    {(X=10)}
```

```
(implies T (equal (+ 0 10) 10))
```

Exactly the condition we expected to get from a VCG.

But we did not implement a VCG!!

Total Correctness

Unfortunately, Moore's method cannot be directly applied to get total correctness.

Recall that we need to prove:

```
(defthm rank-decreases
  (implies (and (inv s)
                (not (exit s)))
           (o< (rank (step s))
               (rank s))))
```

But we do not want to attach ranking functions with every state!

Even if we did, Moore's invariant will likely not help us prove the above theorem.

Characterizing Cutpoints directly

```
(defpun cutsteps-tail (s i)
  (if (cutpoint s) i
    (cutsteps-tail (step s) (+ i 1))))

(defun-sk exists-default ()
  (exists s (not (cutpoint s))))

(defun default () (exists-default-witness))

(defun next-cutpoint (s)
  (if (cutpoint (run s (cutsteps-tail s 0)))
      (run s (cutsteps-tail s 0))
    (default)))
```

Characterizing Correctness Conditions

```
(implies (pre s) (and (cutpoint s) (assertion s)))
(implies (and (exitpoint s) (assertion s)) (post s))
(implies (exitpoint s) (cutpoint s))

(implies (and (cutpoint s)
              (assertion s)
              (not (exitpoint s)))
           ;; Actually you can also assume that
           ;; an exitpoint is reachable from s, but we
           ;; ignore that now.
         (assertion (next-cutpoint (step s))))
```

These four conditions imply partial correctness. The last one is the hard constraint.

Partial Correctness: Restated

```
(defpun exitsteps-tail (s i)
  (if (exitpoint s) i
    (exitsteps-tail (step s) (1+ i))))

(defun exitsteps (s)
  (let ((steps (exitsteps s 0)))
   (if (exitpoint (run s steps)) steps (omega))))

(defthm partial-correctness
  (implies (and (pre s)
                (exitpoint (run s n)))
           (let ((steps (exitsteps s)))
             (and (exitpoint (run s steps))
                  (post (run s steps))))))
```

Total Correctness: Constraints

```
(implies (pre s) (and (cutpoint s) (assertion s)))
(implies (and (exitpoint s) (assertion s)) (post s))
(implies (exitpoint s) (cutpoint s))
(o-p (rank s))

(implies (and (cutpoint s) (assertion s)
              (not (exitpoint s)))
         (let ((ns (next-cutpoint (step s))))
            (and (cutpoint ns)
                 (assertion ns)
                 (o< (rank ns) (rank s)))))
```

Notice that ranking functions are required to be specified only at cutpoints.

Total Correctness: Statement

```
(defthm total-correctness
  (implies (pre s)
           (let ((ns (run s (exitsteps s))))
              (and (exitpoint ns)
                   (post ns)))))
```

Partial (and total) correctness can be proven easily given (corresponding) encapsulated functions `pre`, `post`, `cutpoint`, `assertion`, (and `rank`).

So What?

I have proven the generic (partial and total) correctness theorems, from encapsulated constraints.

But so what? We want to actually use it for proving correctness of real programs.

How do we do that?

Concretizing Generic Proofs

Three key ingredients:

- Functional instantiation
- Symbolic Simulation
- Macros

Functional Instantiation

Suppose you are now given “concrete” functions `prec`, `postc`, etc., and you want to prove the concrete (partial or total) correctness theorem.

```
(defthm concrete-total-correctness
  (implies (prec s0)
           (and (exitpointc (run s0 (exitstepsc s0)))
                (postc (run s0 (exitstepsc s0)))))
  :hints (("Goal" :instance
                      (:functional-instance
                        total-correctness
                        (pre prec)
                        (post postc)
                        ....)
                     (s s0))))
```

Functional Instantiation

Functional instantiation allows you to prove a concrete theorem by instantiating an “abstract” theorem, as long as the concrete functions satisfy the constraints of the abstract function.

Thus functional instantiation requires that you be able to prove first:

```
(implies (prec s) (and (cutpointc s) (assertionc s)))
(implies (exitpointc s) (cutpointc s))
....
```

Functional Instantiation

But wait! What if the functions `prec`, `postc`, etc. take more than one argument?

For example, we might say:

- `(pre k s)`: In state `s`, the memory location `1000` contains a `32`-bit integer `k`.
- `(post k s)`: In state `s` the memory location `1000` contains a `32`-bit integer whose value is `(fix (fib k))`.

At the least we want to allow the user to write such functions.

Functional Instantiation

No problem. You can instantiate a unary abstract function with a concrete function of any arity.

```
(defthm concrete-total-correctness
  (implies (prec s0)
           (and (exitpointc (runc s0 (exitstepsc s0)))
                (postc (run s0 (exitstepsc s0)))))
  :hints (("Goal" :instance
                      (:functional-instance
                        total-correctness
                        (pre (lambda (s) (prec n s)))
                        (post (lambda (s) (postc n s)))
                        ....)
                     (s s0))))
```

Symbolic Simulation

Functional instantiation requires that we prove the constraints on the abstract functions for the concrete ones.

But there was one difficult constraint.

```
(implies (and (cutpointc s)
              (assertionc s)
              (not (exitpointc s)))
         (assertion (next-cutpointc (stepc s))))
```

How will we prove that?

Symbolic Simulation

We prove the following two theorems about `next-cutpointc`. (Again by instantiating the corresponding generic theorems about `next-cutpoint`).

```
(implies (cutpointc s) (equal (next-cutpointc s) s))

(implies (not (cutpointc s))
         (equal (next-cutpointc s)
                (next-cutpointc (stepc s))))
```

These rules allow us to symbolically simulate from cutpoint to cutpoint, much like the way Moore's method did.

Macros: Putting them all together

I will show you a simple macro `defsimulate`, that does this functional instantiation stuff, and thereby proves the concrete correctness theorems.

The TINY fib program

```
100  pushsi 1     *start*
102  dup
103  dup
104  pop 20                      fib0 := 1;
106  pop 21                      fib1 := 1;
108  sub                         n := max(n-1,0);
109  dup          *loop*
110  jumpz 127                   if n == 0, goto *done*;
112  pushs 20
113  dup
115  pushs 21
117  add
118  pop 20                      fib0 := fib0 + fib1;
120  pop 21                      fib1 := fib0 (old value);
122  pushsi 1
```

```
124  sub                             n := max(n-1,0);
125  jump 109                        goto *loop*;
127  pushs 20     *done*
129  add                             return fib0 + n;
130  halt         *halt*
```