Making Induction Manifest in Modular ACL2

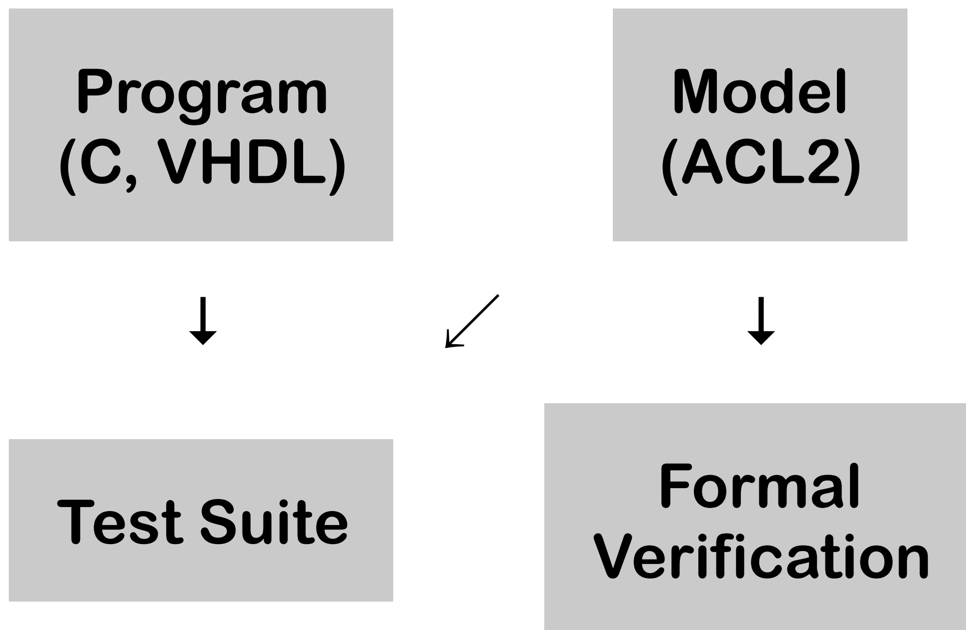
Carl Eastlund Matthias Felleisen

cce@ccs.neu.edu matthias@ccs.neu.edu

Northeastern University

Boston, MA, USA

Program Verification in ACL2


Program
(C, VHDL)
Model
(ACL2)
Test Suite
Formal
Verification

```
(defun setp (s) (no-duplicatesp-equal s))
(defun insert (x s) (add-to-set-eql x s))
```

```
(defthm insert-preserves-setp
  (implies (setp s)
           (setp (insert x s))))
```

Termination Argument (Trivial)?

⇓

```
(defun setp (s) (no-duplicatesp-equal s))
(defun insert (x s) (add-to-set-eql x s))
```

⇓

Rewrite Rule.

Validity?

⇓

```
(defthm insert-preserves-setp
  (implies (setp s)
           (setp (insert x s))))
```

⇓

Rewrite Rule.

```
(defun join (l s)
  (if (endp l)
      s
      (insert (car l) (join (cdr l) s))))
```

```
(defthm join-preserves-setp
  (implies (and (true-listp l) (setp s))
           (setp (join l s))))
```

Termination Argument?

⇓

```
(defun join (l s)
  (if (endp l)
      s
      (insert (car l) (join (cdr l) s))))
```

⇓

Rewrite Rule + Induction Scheme.

Validity by Induction?

⇓

```
(defthm join-preserves-setp
  (implies (and (true-listp l) (setp s))
           (setp (join l s))))
```

⇓

Rewrite Rule.

```lisp
(defun setp (s) (no-duplicatesp-equal s))
(defun insert (x s) (add-to-set-eql x s))

(defthm insert-preserves-setp
  (implies (setp s)
           (setp (insert x s))))


(defun join (l s)
  (if (endp l)
      s
      (insert (car l) (join (cdr l) s))))

(defthm join-preserves-setp
  (implies (and (true-listp l) (setp s))
           (setp (join l s))))
```

```
(defun setp (s) (no-duplicatesp-equal s))
(defun insert (x s) (add-to-set-eql x s))

(defthm insert-preserves-setp
  (implies (setp s)
           (setp (insert x s))))
```

```
(defun join (l s)
  (if (endp l)
      s
      (insert (car l) (join (cdr l) s))))

(defthm join-preserves-setp
  (implies (and (true-listp l) (setp s))
           (setp (join l s))))
```

?

```
(defun join (l s)
  (if (endp l)
      s
      (insert (car l) (join (cdr l) s))))

(defthm join-preserves-setp
  (implies (and (true-listp l) (setp s))
           (setp (join l s))))
```

? ? ?

```
(defun join (l s)
  (if (endp l)
      s
      (insert (car l) (join (cdr l) s))))

(defthm join-preserves-setp
  (implies (and (true-listp l) (setp s))
           (setp (join l s))))
```

Taking a Program Apart

```
(interface Insert
  (sig setp (s))
  (sig insert (x s))

  (con insert-preserves-setp
    (implies (setp s)
             (setp (insert x s)))))

(interface Join
  (extend Insert)

  (sig join (l s))

  (con join-preserves-setp
    (implies (and (true-listp l) (setp s))
             (setp (join l s)))))
```

```
(module JoinMod
  (import Insert)



  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))



  (export Join))
```

```
(module JoinMod
  (import Insert)
```

⇓

Names + Rewrite Rules.

Termination Argument?

⇓

```
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
```

⇓

Rewrite Rule + Induction Scheme.

Validity by Induction?

⇓

```
  (export Join))
```

```
(interface BigStep
  (sig eval (e)) #|contracts|#)

(interface SmallStep
  (sig step (e)) #|contracts|#
  (sig step-all (e)) #|contracts|#)

(interface Equivalence
  (extend BigStep SmallStep)
  (con big-step=small-step
    (implies (expr-p e)
             (equal (eval e) (step-all e)))))
```

```
(module SmallStepMod
  (defun step (e) ...)


  (defun step-all (e)
    (cond ((integerp e) e)
          ((calc-p e) (step-all (step e)))))



  (export SmallStep))
```

```
(module SmallStepMod
  (defun step (e) ...)
```

Termination Argument?

⇓

```
  (defun step-all (e)
    (cond ((integerp e) e)
          ((calc-p e) (step-all (step e)))))
```

⇓

Rewrite Rule + Induction Scheme.

Validity by Induction?

⇓

```
  (export SmallStep))
```

```
(module EquivalenceMod
  (import BigStep SmallStep)









  (export Equivalence))
```

```
(module EquivalenceMod
  (import BigStep SmallStep)
```

⇓

Names + Rewrite Rules.

Validity by Induction?

⇓

```
  (export Equivalence))
```

```
(module EquivalenceMod
  (import BigStep SmallStep)
```

⇓

Names + Rewrite Rules.

Termination Argument?

⇓

```
(defun recursion (e)
  (cond ((integerp e) nil)
        ((calc-p e) (recursion (step e)))))
```

⇓

Rewrite Rule + Induction Scheme.

Validity by Induction?

⇓

```
(export Equivalence))
```

```
(interface BigStep
  (sig eval (e)) #|contracts|#)

(interface SmallStep
  (sig step (e)) #|contracts|#
  (sig step-all (e)) #|contracts|#

                                                                  )

(interface Equivalence
  (extend BigStep SmallStep)
  (con big-step=small-step
    (implies (expr-p e)
             (equal (eval e) (step-all e)))))
```

```
(interface BigStep
  (sig eval (e)) #|contracts|#)

(interface SmallStep
  (sig step (e)) #|contracts|#
  (sig step-all (e)) #|contracts|#
  (fun recursion (e)
    (cond ((integerp e) nil)
          ((calc-p e) (recursion (step e))))))

(interface Equivalence
  (extend BigStep SmallStep)
  (con big-step=small-step
    (implies (expr-p e)
             (equal (eval e) (step-all e)))))
```

```
(interface BigStep
  (sig eval (e)) #|contracts|#)

(interface SmallStep
  (sig step (e)) #|contracts|#
  (fun step-all (e)
    (cond ((integerp e) e)
          ((calc-p e) (step-all (step e)))))
                                                    )

(interface Equivalence
  (extend BigStep SmallStep)
  (con big-step=small-step
    (implies (expr-p e)
             (equal (eval e) (step-all e)))))
```

```
(module SmallStepMod
  (defun step (e) ...)
```

Validity and Termination Argument?

⇓

```
  (export SmallStep)
```

⇓

Names, Rewrite Rules, and Induction Scheme.)

```
(module EquivalenceMod
  (import BigStep SmallStep)
```

$\Downarrow$

Names, Rewrite Rules, and Induction Scheme.

Validity by Induction?

$\Downarrow$

```
  (export Equivalence))
```

```
(defun D (x) d)
(defthm E e)
(defun F (y) f)
(defthm G g)
(defun H (z) h)
(defthm I i)
```

```
(defun D (x) d)
(defthm E e)


(defun F (y) f)
(defthm G g)


(defun H (z) h)
(defthm I i)
```

```
(interface A
  (defun D (x) d)
  (defthm E e))

(interface B
  (extend A)
  (defun F (y) f)
  (defthm G g))

(interface C
  (extend A B)
  (defun H (z) h)
  (defthm I i))
```

```
(interface A
  (fun    D (x) d)
  (defthm E e))

(interface B
  (extend A)
  (fun    F (y) f)
  (defthm G g))

(interface C
  (extend A B)
  (fun    H (z) h)
  (defthm I i))
```

```
(interface A
  (fun    D  (x)  d)
  (con     E  e))

(interface B
  (extend A)
  (fun    F  (y)  f)
  (con     G  g))

(interface C
  (extend A B)
  (fun    H  (z)  h)
  (con     I  i))
```

```
(interface A
  (fun    D  (x)  d)
  (con     E  e))

(interface B
  (extend A)
  (fun    F  (y)  f)
  (con     G  g))

(interface C
  (extend A B)
  (fun    H  (z)  h)
  (con     I  i))
```

```
(module M
  (export A))

(module N
  (import A)
  (export B))

(module O
  (import A B)
  (export C))
```

Lemma	Modular	ACL2	Optimized
`random/type`	0.05s	0.05s	0.05s
`tick/type`	0.01s	142.88s	2.00s
`tick/in-bounds`	0.01s	136.67s	2.28s
`tick/uncrossed`	0.02s	320.84s	2.29s

Putting a Program Back Together

```
(link InsertJoinMod
  (InsertMod JoinMod))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)
  (import Insert)
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)
  (import Insert)
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)
  (import Insert)
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)
  (import Insert)
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)
  (import Insert)
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)
  ~~(import Insert)~~
  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module InsertJoinMod
  (defun setp (s) (no-duplicatesp-equal s))
  (defun insert (x s) (add-to-set-eql x s))
  (export Insert)

  (defun join (l s)
    (if (endp l)
        s
        (insert (car l) (join (cdr l) s))))
  (export Join))

(invoke InsertJoinMod)

(join (list 1 2 3) (list 2 3 4))
```

```
(module M
  (export I))
```

```
(module N
  (import I)
  (export J))
```

```
(module M
  (export I))

+

(module N
  (import I)
  (export J))

=

(link MN
  (M N))
```

```
(module M
  (export I))
```

+

```
(module N
  (import I)
  (export J))
```

=

```
(module MN
  (export I)
  (export J))
```

```
(module M          (module N           (module MN
  (export I))   +    (import I)    =     (export I)
                     (export J))         (export J))
```

```
I
```

```
(module M          (module N            (module MN
  (export I))   +    (import I)   =       (export I)
                     (export J))          (export J))
```

$I \quad , \quad I \Rightarrow J$

```
(module M
  (export I))
```

+

```
(module N
  (import I)
  (export J))
```

=

```
(module MN
  (export I)
  (export J))
```

$$\mathbf{I} \quad , \quad \mathbf{I} \Rightarrow \mathbf{J} \quad \vdash \quad \mathbf{I} \wedge \mathbf{J}$$

Program	Modular	ACL2
Worm	135.40s	134.77s
Interpreter	116.37s	115.67s
Graph (DFS/NLG)	9.00s	9.03s
Graph (DFS/ELG)	13.88s	13.82s
Graph (BFS/NLG)	158.11s	158.19s
Graph (BFS/ELG)	445.15s	444.28s

Modular ACL2:

sound,

expressive,

and efficient.

Thank You

Modular ACL2:
`http://www.ccs.neu.edu/~cce/acl2/`