Session 03 — 1:30-2:30, May 13

Teaching Software Correctness

May 13-15, 2008, University of Oklahoma

http://www.cs.ou.edu/~rlpage/SEcollab/tsc

Rex Page, U Oklahoma    page@ou.edu

Assistants

Carl Eastlund (lead), Northeastern U    cce@ccs.neu.edu
Ryan Ralston, U Oklahoma    strawdog@ou.edu
Zac White, U Oklahoma    zacwhite@gmail.com

Collaboration with Matthias Felleisen - NSF/DUE 0633664, 0813529, 0632872

TSC Workshop, May 2008, U Oklahoma
The Method

using lemmas to guide ACL2

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - (avg xs) = average of xs, where xs is a sequence of numbers
    - (var xs) = variance of xs, where xs is a sequence of numbers
  - Verify these properties
    - (avg xs) is a rational number
    - (var xs) is a non-negative, rational number

- Function definitions
  - Average = sum / number of numbers in the list
    - (len xs) = number of numbers in xs (an ACL2 intrinsic)
    - (sum xs) = sum of numbers in xs (not intrinsic, need def'n)
  - Variance = average squared difference, $(x - \mu)^2$
    - Where $x$ comes from $xs$, $\mu = (avg \ xs)$
    - So, can use avg in definition of
Average

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - \((\text{avg } xs) = \text{average of } xs\), where \(xs\) is a sequence of numbers
    - \((\text{var } xs) = \text{variance of } xs\), where \(xs\) is a sequence of numbers
  - Verify these properties
    - \((\text{avg } xs)\) is a rational number
    - \((\text{var } xs)\) is a non-negative, rational number

- Definition of avg

  \[
  \begin{align*}
  \text{(defun sum (xs)} & \text{ (if (endp xs)} \\
  & \quad 0 \\
  & \quad (+ (\text{car } xs) (\text{sum (cadr } xs))) \\
  \text{(defun avg (xs)} & \text{ (/ (sum xs) (len } xs)))
  \end{align*}
  \]

  (sum nil) = 0
  (sum (list x1 x2 ...)) = (+ x1 (sum (list x1 x2 ...))
  (avg xs) = (sum xs)/(len xs)
Average

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - \((\text{avg } x) = \text{average of } x, \text{ where } x \text{ is a sequence of numbers}\)
    - \((\text{var } x) = \text{variance of } x, \text{ where } x \text{ is a sequence of numbers}\)
  - Verify these properties
    - \((\text{avg } x) \text{ is a rational number}\)
    - \((\text{var } x) \text{ is a non-negative, rational number}\)

- Definition of avg
  - \((\text{sum } (xs)) = \text{if } (\text{endp } x) \text{ then } 0 \text{ else } (+ (\text{car } x) (\text{sum } (\text{cdr } x)))\)
  - \((\text{defun avg } x) = (/ (\text{sum } x) (\text{len } x)))\)
  - \((\text{defthm avg-is-rational})\)
  - \((\text{rationalp } (\text{avg } x)))\)

- Properties sum must satisfy (axioms)
  - \((\text{sum nil}) = 0\)
  - \((\text{sum } (\text{list } x_1 x_2 ...)) = (+ x_1 (\text{sum } (\text{list } x_1 x_2 ...)))\)
  - \((\text{avg } x) = (\text{sum } x)/(\text{len } x)\)

- Property we want to derive
  - \((\text{avg } x) \text{ is rational}\)
Average

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - \( (\text{avg } x) = \text{average of } x \), where \( x \) is a sequence of numbers
    - \( (\text{var } x) = \text{variance of } x \), where \( x \) is a sequence of numbers
  - Verify these properties
    - \( (\text{avg } x) \) is a rational number
    - \( (\text{var } x) \) is a non-negative, rational number

- Definition of avg

  \[
  (\text{defun} \text{ sum} (x) \\
  \quad (\text{if} \ (\text{endp} \ x) \\
  \quad \quad 0) \\
  \quad (+ \ (\text{car} \ x) \ (\text{sum} \ (\text{cdr} \ x)))
  \]

  \[
  (\text{defun} \text{ avg} (x) \\
  \quad (/ \ (\text{sum} \ x) \ (\text{len} \ x)))
  \]

  \[
  (\text{defthm} \text{ avg-is-rational} \\
  \quad (\text{rationalp} \ (\text{avg} \ x)))
  \]

Let's propose it to ACL2
Average

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - (avg xs) = average of xs, where xs is a sequence of numbers
    - (var xs) = variance of xs, where xs is a sequence of numbers
  - Verify these properties
    - (avg xs) is a rational number
    - (var xs) is a non-negative, rational number

Definition of avg

(defun sum (xs)
  (if (endp xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(defun avg (xs)
  (/ (sum xs) (len xs)))

(defun avg-is-rational
  (defthm avg-is-rational
    (rationalp (avg xs))))

Eh?

Subgoal *1/2'''
(IMPLIES (AND (CONSP XS)
               (STRINGP (CDR XS))
               (NOT (STRINGP XS))
               (ACL2-NUMBERP (CAR XS))
               (RATIONALP (CAR XS)))
  (********** FAILED **********
Average

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - (avg xs) = average of xs, where xs is a sequence of numbers
    - (var xs) = variance of xs, where xs is a sequence of numbers
  - Verify these properties
    - (avg xs) is a rational number
    - (var xs) is a non-negative, rational number

- Definition of avg
  (defun sum (xs)
   (if (endp xs)
       0
       (+ (car xs) (sum (cdr xs))))))
  (defun avg (xs)
   (/ (sum xs) (len xs))))
  (defthm avg-is-rational
   (implies (rational-listp xs)
            (rationalp (avg xs))))

Eh?
Average

- Project - numeric domain (since it's familiar territory)
  - Define avg and var
    - \((\text{avg } x\text{s})\) = average of \(x\text{s}\), where \(x\text{s}\) is a sequence of numbers
    - \((\text{var } x\text{s})\) = variance of \(x\text{s}\), where \(x\text{s}\) is a sequence of numbers
  - Verify these properties
    - \((\text{avg } x\text{s})\) is a rational number
    - \((\text{var } x\text{s})\) is a non-negative, rational number

- Definition of avg

\[
\text{(defun sum (xs)}
\begin{align*}
&= \text{if (endp } x\text{s) } 0 \\
&\quad + (\text{car } x\text{s}) (\text{sum (cdr } x\text{s)})))
\end{align*}
\]

\[
\text{(defun avg (xs)}
\begin{align*}
&= \text{(/ (sum } x\text{s) (len } x\text{s)}))
\end{align*}
\]

\[
\text{(defthm avg-is-rational}
\begin{align*}
&= \text{(implies (rational-listp } x\text{s) (rationalp (avg } x\text{s)})))
\end{align*}
\]

Eh? Let's give ACL2 another go
**Variance**

- Variance = average squared difference, \((x - \mu)^2\)
  - Where \(x\) comes from \(xs\), \(\mu = (\text{avg} \; xs)\)
  - \(\text{var} \; xs = \text{avg}((\text{list} \; x_1 \; x_2 \; ...) - \mu)^2\)
  - \((\text{list} \; y_1 \; y_2 \; ...)^2 = (\text{list} \; y_1^2 \; y_2^2 \; ...))\)
Variance

Variance = average squared difference, $(x - \mu)^2$
- Where $x$ comes from $xs$, $\mu = (\text{avg } xs)$
- $\text{var } xs = \text{avg}( (\text{list } x_1 x_2 ...) - \mu )^2$

Vector minus scalar, vector squared

(defun v-s (xs s)
  (if (endp xs)
      nil ; (v-s nil) = nil
      (cons (- (car xs) s) ; (v-s (list x_1 x_2 ...)) =
        (v-s (cdr xs) s)))) ; (cons (- x_1 s) (v-s (list x_2 ...)))

(defun v^2 (xs)
  (if (endp xs)
      nil ; (v^2 nil) = nil
      (cons (* (car xs) (car xs)) ; (v^2 (list x_1 x_2 ...)) =
        (v^2 (cdr xs)))) ; (cons (* x_1 x_1) (v^2 (list x_2 ...)))

Variance

- Variance = average squared difference, \((x - \mu)^2\)
  - Where \(x\) comes from \(xs\), \(\mu = (\text{avg } xs)\)
  - So, can use \text{avg} in definition of

Vector minus scalar, vector squared

```lisp
(defun v-s (xs s)
  (if (endp xs)
      nil
    (cons (- (car xs) s)
      (v-s (cdr xs) s)))))

(defun v^2 (xs)
  (if (endp xs)
      nil
    (cons (* (car xs) (car xs))
      (v^2 (cdr xs)))))

(defun var (xs)
  (avg (v^2 xs (v-s) xs (avg xs))))
```
Av<sub>g</sub>-Var Book

packaging collections of functions or theorems

(in-package "ACL2") ← importable module

(defun sum (xs)
  (if (null xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(defun avg (xs)
  (/ (sum xs) (len xs)))

(defun v−s (xs  s)
  (if (endp xs)
      nil ; (v−s nil) = nil
      (cons (- (car xs) s) ; (v−s (list x<sub>1</sub> x<sub>2</sub> ...)) =
              (v−s (cdr xs) s)))) ;  (cons (− x<sub>1</sub> s) (v−s (list x<sub>2</sub> ...))

(defun v^2 (xs)
  (if (endp xs)
      nil ; (v^2 nil) = nil
      (cons (* (car xs) (car xs)) ; (v^2 (list x<sub>1</sub> x<sub>2</sub> ...)) =
              (v^2 (cdr xs)))))) ;   (cons (* x<sub>1</sub> x<sub>2</sub>)(v^2 (list x<sub>2</sub> ...)))

(defun var (xs)
  (avg (v^2 xs (avg xs))))

Save as "avg-var.lisp"

Import to other modules: (include-book "avg-var.lisp")

For DrACuLa purposes:

save avg-var.lisp in same directory as module that imports it (or sub-directory)

; (v−s nil) = nil
; (v−s (list x<sub>1</sub> x<sub>2</sub> ...)) =
; (cons (− x<sub>1</sub> s) (v−s (list x<sub>2</sub> ...))

; (v^2 nil) = nil
; (v^2 (list x<sub>1</sub> x<sub>2</sub> ...)) =
; (cons (* x<sub>1</sub> x<sub>2</sub>)(v^2 (list x<sub>2</sub> ...)))

; var xs = avg((xs - (avg xs))^2)
Variance - Derived Properties

(defun v-s (xs s)
  (if (endp xs)
      nil
      (cons (- (car xs) s)
            (v-s (cdr xs) s)))))

(defun v^2 (xs)
  (if (endp xs)
      nil
      (cons (* (car xs) (car xs))
            (v^2 (cdr xs)))))

(defun var (xs)
  (avg (v^2 xs (avg xs))))

(defthm var-is-rational
  (implies (rational-listp xs)
           (rationalp (var xs))))

; (v-s nil) = nil
; (v-s (list x_1 x_2 ...)) =
; (cons (- x_1 s) (v-s (list x_2 ...)))

; (v^2 nil) = nil
; (v^2 (list x_1 x_2 ...)) =
; (cons (* x_1 x_2)(v^2 (list x_2 ...)))
; var xs = avg((xs - (avg xs))^2)

TSC Workshop, May 2008, U Oklahoma
**Variance - Derived Properties**

\[
\text{(defun v-s (xs s)} \\
\quad \text{(if (endp xs)} \\
\quad \quad \text{nil)} \\
\quad \quad \text{(cons (-(car xs) s)} \\
\quad \quad \quad \text{(v-s (cdr xs) s))})
\]

\[
\text{(defun v^2 (xs)} \\
\quad \text{(if (endp xs)} \\
\quad \quad \text{nil)} \\
\quad \quad \text{(cons (* (car xs) (car xs)} \\
\quad \quad \quad \text{(v^2 (cdr xs)))})
\]

\[
\text{(defun var (xs)} \\
\quad \text{(avg (v^2 xs (avg xs)))}
\]

**derived properties of var**

\[
\text{(defthm var-is-rational} \\
\quad \text{(implies (rational-listp xs)} \\
\quad \quad \text{(rationalp (var xs))})
\]

\[
\text{(defthm var-is-positive} \\
\quad \text{(implies (rational-listp xs)} \\
\quad \quad \text{(>= (var xs) 0))})
\]

**derived prop of avg**

\[
\text{(defthm avg-is-rational} \\
\quad \text{(implies (rational-listp xs)} \\
\quad \quad \text{(rationalp (avg xs))})
\]
Variance - Derived Properties

(defun v-s (xs s)  
  (if (endp xs)  
      nil  
      (cons (- (car xs) s)  
            (v-s (cdr xs) s))))

(defun v^2 (xs)  
  (if (endp xs)  
      nil  
      (cons (* (car xs) (car xs))  
            (v^2 (cdr xs)))))

(defun var (xs)  
  (avg (v^2 xs (avg xs))))

; (v-s nil) = nil  
; (v-s (list x₁ x₂ ...)) =  
; (cons (- x₁ s) (v-s (list x₂ ...)))

; (v^2 nil) = nil  
; (v^2 (list x₁ x₂ ...)) =  
; (cons (* x₁ x₂)(v^2 (list x₂ ...)))  
; var xs = avg((xs - (avg xs))^2)

derived properties of var

(defthm avg-is-rational  
  (implies (rational-listp xs)  
            (rationalp (var xs))))

(defthm avg-is-positive  
  (implies (rational-listp xs)  
            (> (var xs) 0)))

derived prop of avg

(let ((x (list 1 2 3)))  
  (avg (v^2 x (avg x))))

Let's propose it to ACL2

May 2008, U Oklahoma
Variance - Derived Properties

(defun v-s (xs s)
  (if (endp xs)
      nil
      (cons (- (car xs) s) ; (v-s (list x1 x2 ...)) =
             (v-s (cdr xs) s)))))
(defun v^2 (xs)
  (if (endp xs)
      nil
      (cons (* (car xs) (car xs)) ; (v^2 (list x1 x2 ...)) =
             (v^2 (cdr xs)))))
(defun var (xs)
  (avg (v^2 xs (avg xs)))))

Let's propose it to ACL2

Yikes! Let's go into high gear
Where the proof goes wrong

- What we’re trying to prove
  (defthm var-is-positive
   (implies (rational-listp xs)
    (>= (var xs) 0)))

- Proof generates this odd subgoal
  Subgoal *1.1/1''
  (IMPLIES (RATIONALP XS1)
    (<= 0
     (AVG (V^2 (V-S (LIST XS1) (AVG (LIST XS1))))))).

- Why is it worrying about with a one-element list?
  - Because of this:
    (defthm avg-of-pos-is-pos
     (implies (and (true-listp xs)
                   (rational-listp xs)
                   (not (null xs))
                   (positive-listp xs))
      (>= (avg xs) 0)))

Unnecessary condition - thm is too specialized
More gen’l thm easier
Keep on Truckin’

another little project

- Compress — remove identical adjacent list elements
  - \((\text{compress (list 1 1 2 2 2 3 4 4 5 1 1)})) = (\text{list 1 2 3 4 5 1})\)

- Some simple properties of the compress function
  - \((\text{compress nil}) = \text{nil}\)
  - \((\text{compress (list x)}) = (\text{list x})\)
  - \((\text{compress (list } x_1 \ x_2 \ x_3 \ ...))\)
    - \(x_1 = x_2\)
    - \(x_1 \neq x_2\)

\((\text{compress (list } x_2 \ x_3 \ ...))\) \(= (\text{cons } x_1 \ (\text{compress (list } x_2 \ x_3 \ ...)))\)

-defun compress (xs)
  (if (endp (cdr xs))
    xs
    (if (equal (car xs) (cadr xs))
      (compress (cdr xs))
      (cons (car xs) (compress (cdr xs))))))

A list that’s empty with its first element removed:
\((\text{null (cdr xs)})\)

0 or 1 elements:
- compressed list identical to input
- 0 or 1 element
- 1st two elems same
- 1st two elems differ
Derived Properties for Compress
(compress (list 1 1 2 2 2 3 4 4 5 1 1)) = (list 1 2 3 4 5 1)

- **Correctness properties**
  - Compress delivers true-list
  - Conservation of values
    - Values that were in list, remain
    - No new values appear
  - No adjacent duplicates

  We've seen these before
  ... Goes thru easily
  Goes thru easily
  ... but worth looking stating
  This is the hard part

- **Other derived properties important in some contexts**
  - Compress delivers a value no longer than its argument
  - Compress delivers a value shorter than an arg with adjacent dups

- Some of these are harder to derive than others
Conservation of Values

- Values in list before compression remain in list after compression, and vice versa

(defthm compress-conserves-elements
  (implies (true-listp xs)
    (iff (member-equal x xs)
      (member-equal x (compress xs))))))
Adjacent Duplicates Predicate

Inductive definition

(defun adjacent-duplicatep (xs)
  (and (not (endp (cdr xs)))
       (or (equal (car xs) (cadr xs))
           (adjacent-duplicatep (cdr xs))))))

How do we know this definition is right?

- Part of the story
  (defthm no-adj-dups-implies-no-two-adj-elems-same
    (implies (and (true-listp xs)
                   (not (adjacent-duplicatep xs))
                   (natp n)
                   (< n (1- (len xs))))
               (not (equal (Nth n xs)
                           (Nth (1+ n) xs)))))

- Converse is other part of story
Compressed List Has No Adjacent Dups

- **Statement of theorem**
  (defthm compressed-list-has-no-adjacent-duplicates
   (implies (true-listp xs)
                (not (adjacent-duplicatep (compress xs))))))

- **Doesn't go through without help**
  - **Lemma: inductive case**
    (defthm compressed-list-has-no-adj-dups-inductive-case
     (implies (and (true-listp xs)
                   (not (adjacent-duplicatep (compress (cdr xs))))
                   (not (adjacent-duplicatep (compress xs))))))

  **Excerpt from proof attempt**
  Subgoal *1.1/2
  (IMPLIES (AND (CONSP XS4)
                 (NOT (CONSP (CDR XS4)))
                 (TRUE-LISTP XS4)
                 (NOT (EQUAL L1 (CAR XS4)))
                 (NOT (TRUE-LISTP (CONS (CAR XS4) XS4))))))

  **Proof of lemma needs help, too**
  (defthm 1st-elem-is-1st-compressed-elem
   (implies (true-listp xs)
            (equal (car xs)
                   (car (compress xs)))))

  **problems with first element**
  **Lemma for lemma goes through**
**Force Use of Lemma**

**use-instance hint**

- **Use-lemma directive for inductive case**
  ```lisp
  (defthm compressed-list-has-no-adj-dups-inductive-case ;
   lemma
   (implies (and (true-listp xs)
     (not (adjacent-duplicatep (compress (cdr xs)))))
    (not (adjacent-duplicatep (compress xs))))
  :hints
   (("Goal"
     :use (:instance 1st-elem-is-1st-compressed-elem
        (xs (cdr xs)))))
  ```

Excerpt from proof attempt

ACL2 Warning [Use] in
  (DEFTHM COMPRESSED-LIST-HAS-NO-ADJ-DUPS-INDUCTIVE-CASE ...):
It is unusual to :USE an enabled :REWRITE or :DEFINITION rule, so you may want to consider disabling
  (:REWRITE |1ST-ELEM-IS-1ST-COMPRESSED-ELEM|).

Must be using it too much, now
Disable at lower levels

Substitute (cdr xs) from theorem for xs in lemma
Disable Lemma at Lower Levels in Proof

disable, in theory

- Use-lemma directive for inductive case

(defun compressed-list-has-no-adj-dups-inductive-case ;
  (implies (and (true-listp xs)
                (not (adjacent-duplicatep (compress (cdr xs))))))
  (not (adjacent-duplicatep (compress xs))))

:hints
  ("Goal"
   :use (:instance 1st-elem-is-1st-compressed-elem
        (xs (cdr xs)))
   :in-theory (disable 1st-elem-is-1st-compressed-elem))

Use at top level
Disable at lower levels

(defun 1st-elem-is-1st-compressed-elem ;
  (implies (true-listp xs)
            (equal (car xs)
                   (car (compress xs)))))

Substitute (cdr xs) from theorem for xs in lemma

Now, inductive-case Lemma goes through
And, so does theorem
Is That All There Is?
What about big programs?

- What was the first program you wrote?
  - Hello World
- Learning to use a tool like ACL2 takes a similar amount of effort as learning to use a language like Java or C or Lisp
- Lots of software components dist’d with ACL2
  - :System “books”
  - ACL2 workshop books - a growing resource
- Software components critical to applications are good targets for analysis through ACL2
- Increment of effort in software development
  - Learning curve - less steep than widely believed
  - Rough guess - doubles engineering time
Learning to Use ACL2
Kaufmann/Moore/Maniolas Textbook
ACL2 Website - just Google ACL2

- Computer-Aided Reasoning - An Approach
  - Excellent explanations of deep, important material
- ACL2 Website
  - Tutorials, examples, wealth of information
- Online help
  - World-class experts reachable by email
  - Great response times, great help
- Teaching Software Correctness website
  - Teaching/Learning resource — a useful one, we hope!
The End