Property-Based Testing
a catalog of classroom examples

Rex Page
University of Oklahoma

supported by CCF program of NSF

Knowing What It Does

- Dialogue
  - Socrates: How do you know what your software does?
  - Engineer: I test it.
  - Socrates: How do you test it?
  - Engineer: I think of things that might happen and test them.
  - Socrates: How many tests?
  - Engineer: About four. Maybe five. Two or three, anyway.
  - Socrates: That about covers it?
  - Engineer: Yeah, I check it out pretty well.
  - Socrates: How about testing all the cases?
  - Engineer: Well, maybe for really complicated programs.
  - Socrates: How many tests then?
  - Engineer: A lot … hundreds for sure.

What to Do?

- A program is a formula in a formal system
  - It has a precise meaning
  - Reasoning about its meaning is an application of logic
- Functional programs are especially attractive
  - Ordinary, algebraic reasoning based on equations
  - Classical logic
    - ✓ Not exotic variants like temporal logic, modal logic, …

Programs = Axiomatic Equations

- A program is a formula in a formal system
  - Its meaning can be specified precisely
  - So, reasoning about its meaning is an application of logic
- Functional programs are especially attractive
  - Ordinary, algebraic reasoning based on equations
  - Classical logic
    - ✓ Not exotic variants like temporal logic, modal logic, …
- Functional program = set of equations {axioms}

```
(first (cons x xs)) = x                               {first}
(rest  (cons x xs)) = xs                              {rest}
(cons x0 (x1 x2 … xn)) = (x0 x1 x2 … xn)              {cons}
(append nil ys) = ys                                  {app0}
(append (cons x xs) ys) = (cons x (append xs ys))     {app1}
```

- Criteria for defining operations
  - Consistent, Comprehensive, Computational *{the 3 C's}*

What about Tests?

- Functional program = set of equations {axioms}

```
(first (cons x xs)) = x                                      {first}
(rest  (cons x xs)) = xs                                     {rest}
(cons x0 (x1 x2 … xn)) = (x0 x1 x2 … xn)                     {cons}
(append nil ys) = ys                                         {app0}
(append (cons x xs) ys) = (cons x (append xs ys))            {app1}
```

- Test = Boolean formula expressing expectation
  - Derivable (the programmer hopes) from the program {axioms}

```
(append xs (append ys zs)) = (append (append xs ys) zs) {assoc}
```

Programs vs Tests

- Functional program = set of equations {axioms}

```
(append nil ys) = ys                                    {app0}
(append (cons x xs) ys) = (cons x (append xs ys))       {app1}
```

- Test = Boolean formula expressing expectation
  - Derivable (the programmer hopes) from the program {axioms}

```
(append xs (append ys zs)) = (append (append xs ys) zs) {assoc}
```

- Program = Equations = Tests
  - Programs and tests are based on <u>the same idea</u> (equations)
  - Program

```
(append nil ys) = ys                                ; app0
(append (cons x xs) ys) = (cons x (append xs ys))   ; app1
```

  - Test

```
(append xs (append ys zs)) = (append (append xs ys) zs) ; assoc
```

Program = Tests

- Functional program = set of equations {axioms}

```
(append nil ys) = ys                                    {app0}
(append (cons x xs) ys) = (cons x (append xs ys))       {app1}
```

- Test = Boolean formula expressing expectation
  - Derivable (the programmer hopes) from the program {axioms}

```
(append xs (append ys zs)) = (append (append xs ys) zs) {assoc}
```

- Program: axiomatic equations

ACL2 function definition

```
(defun append (xs ys)
  (if (consp xs)
      (cons (first xs) (append (rest xs) ys))        ; app1
      ys)                                            ; app0
```

- Tests: derivable equations

Dracula automated testing

```
(defproperty append-associative
  (xs :value (random-list-of (random-symbol))
   ys :value (random-list-of (random-symbol))
   zs :value (random-list-of (random-symbol)))
  (equal (append xs (append ys zs))                  ; assoc
         (append (append xs ys) zs)))
```

Hughes Property Categories

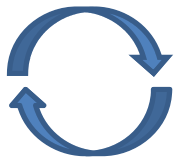
- Comparing results from two ways of doing something
  - (one-way x) = (other-way x)
  - It's nice if one way is "obviously correct"
  - Even if it's not, checking it from two angles helps
- Checking that one function inverts another
  - (decode (encode x)) = x
  - Uncommon to make consistent errors both ways

Hughes Property Categories

- Comparing results from two ways of doing something
  - (one-way x) = (other-way x)
  - It's nice if one way is "obviously correct"
  - Even if it's not, checking it from two angles helps

commuting diagram

- Checking that one function inverts another
  - (decode (encode x)) = x
  - Uncommon to make consistent errors both ways



round trip

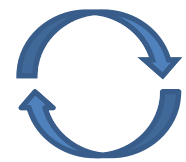
- Useful properties often fall into one of these types
  - An observation from experience of John Hughes
  - Categories help programmers conjure up good tests

Hughes Property Categories

- Comparing results from two ways of doing something
  - (one-way x) = (other-way x)
  - It's nice if one way is "obviously correct"
  - Even if it's not, checking it from two angles helps

commuting diagram

- Checking that one function inverts another
  - (decode (encode x)) = x
  - Uncommon to make consistent errors both ways



round trip

- Useful properties often fall into one of these types
  - An observation from experience of John Hughes
  - Categories help programmers conjure up good tests
- Same categories in classroom examples?
  - Software properties from a decade of courses at OU

Informal Specs and Properties

- Informal specifications of some list operators

  (append (x_1 x_2 … x_m) (y_1 y_2 … y_n)) = (x_1 x_2 … x_m y_1 y_2 … y_n)
  (prefix n (x_1 x_2 … x_n x_{n+1} x_{n+2} …) = (x_1 x_2 … x_n)
  (suffix n (x_1 x_2 … x_n x_{n+1} x_{n+2} …) = (x_{n+1} x_{n+2} …)

- Some equations the operators satisfy in well-chosen cases

Axiomatic Properties

✓Consistent, Comprehensive, Computational

definitions

```
(append nil ys) = ys                                ; app0
(append (cons x xs) ys) = (cons x (append xs ys))   ; app1
(prefix 0 xs)  = nil                                ; pfx0a
(prefix n nil) = nil                                ; pfx0b
(prefix (+ n 1) (cons x xs))= (cons x (prefix n xs)) ; pfx1
(suffix 0 xs)  = nil                                ; sfx0
(suffix (+ n 1) (cons x xs))= (suffix n xs))        ; sfx1
```

- Some other equations we expect the operators satisfy

```
(append xs (append ys zs)) = (append (append xs ys) zs) ; assoc
(prefix (len xs) (append xs ys)) = xs                   ; app-pfx
(suffix (len xs) (append xs ys)) = ys                   ; app-sfx
```

tests

Derived Properties

ACL2 Syntax for Those Equations

- Axiomatic properties

```
(defun append (xs ys)
  (if (consp xs)
      (cons (first xs) (append (rest xs) ys))        ; app1
      ys))                                           ; app0
(defun prefix (n xs)
  (if (and (posp n) (consp xs))
      (cons (first xs) (prefix (- n 1) (rest xs)))   ; pfx1
      nil))                                          ; pfx0
(defun suffix (n xs)
  (if (posp n)
      (suffix (- n 1) (rest xs))                     ; sfx1
      xs))                                           ; sfx0
```

definitions

- Derived properties for testing or verification

```
(defthm app-assoc
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
(defthm app-pfx
  (implies (true-listp xs)
           (equal (prefix (len xs) (append xs ys)) xs)))
(defthm app-sfx
  (equal (suffix (len xs) (append xs ys)) ys))
```

tests / theorems (mechanized logic)

Theorem = Property

without :value, "implies" for ":where"

- Axiomatic properties

```
(defun append (xs ys)
  (if (consp xs)
      (cons (first xs) (append (rest xs) ys))        ; app1
      ys))                                           ; app0
(defun prefix (n xs)
  (if (and (posp n) (consp xs))
      (cons (first xs) (prefix (- n 1) (rest xs)))   ; pfx1
      nil))                                          ; pfx0
(defun suffix (n xs)
  (if (posp n)
      (suffix (- n 1) (rest xs))                     ; sfx1
      xs))                                           ; sfx0
```

definitions

- Derived properties for testing or verification

theorem

```
(defthm app-pfx
  (implies (true-listp xs)
           (equal (prefix (len xs) (append xs ys)) xs)))
(defproperty app-pfx-as-property
  (xs :value (random-list-of (random-symbol))
      :where (true-listp xs))
  (equal (prefix (len xs) (append xs ys)) xs))
```

property

More Properties

- Additional derived properties of append, prefix, suffix

```
(defthm app-preserves-len
  (equal (len (append xs ys))
         (+ (len xs) (len ys))))
(defthm app-conserves-elements
  (iff (member-equal a (append xs ys))
       (or (member-equal a xs) (member-equal a ys))))
(defthm pfx-len
  (implies (natp n) (<= (len (prefix n xs)) n)))
(defthm sfx-len
  (implies (natp n) (<= (len (suffix n xs))
                        (max 0 (- (len xs) n)))))
```

tests / theorems (mechanized logic)

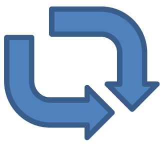
- Derived properties for testing or verification

```
(defthm app-assoc
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
(defthm app-pfx
  (implies (true-listp xs)
           (equal (prefix (len xs) (append xs ys)) xs)))
(defthm app-sfx
  (equal (suffix (len xs) (append xs ys)) ys))
```

tests / theorems (mechanized logic)

Typical Classroom Examples

❑ Commuting diagram properties



- Append preserves length and conserves elements
- Law of added exponents: $x^m x^n = x^{m+n}$
- Russian peasant exponentiation: $x^n = x \cdot x \cdot \ldots\ x = x^{\lfloor n/2 \rfloor} x^{n \bmod 2}$
- Scalar times vector: $s \cdot x_k$ = k^{th} element of $s \cdot [x_1, x_2, \ldots\ x_n]$
- Nested recursion vs tail recursion (*eg*, list-reversal, Fibonacci)
- Arithmetic on numerals

  (numb(add (bits *x*) (bits *y*))) = $x + y$

  (numb(mul (bits *x*) (bits *y*))) = $x \cdot y$

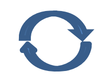
  (low-order-bit (bits(2·*x*))) = 0

  (numb(insert-high-order-bits *n* (bits *x*))) = $x \cdot 2^n$

Property Counts

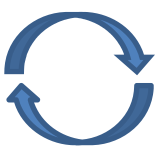
from SE lectures

26 

23 

22 others

❑ Round-trip properties



- Double reverse: (reverse (reverse *xs*)) = *xs*
- Division check: *y*·(div *x y*) + (mod *x y*) = *x*
- Multiplex, demultiplex: (mux (dmx *xs*)) = *xs*, (dmx (mux *xs ys*)) = (*xs ys*)
- Concatenate prefix/suffix: (append (prefix *n xs*) (suffix *n xs*)) = *xs*
- Linear encryption: (decrypt (encrypt *msg*)) = *msg*
- Convert number to numeral and back: (numb (bits *x*)) = *x*

Linear Encryption

add adjacent codes, mod code-space size

```
(defun encrypt-pair (m x x-nxt)
  (mod (+ x x-nxt) m))
(defun decrypt-pair (m x-encrypted y-decrypted)
  (mod (- x-encrypted y-decrypted) m))
(defun encrypt (m xs)
  (if (consp (cdr xs))
      (cons (encrypt-pair m (car xs) (cadr xs))
            (encrypt m (cdr xs)))
      (list (encrypt-pair m (car xs) (1- m)))))
(defun decrypt (m ys)
  (if (consp (cdr ys))
      (let* ((decrypted-cdr (decrypt m (cdr ys))))
        (cons (decrypt-pair m (car ys) (car decrypted-cdr))
              decrypted-cdr))
      (list (decrypt-pair m (car ys) (1- m)))))
```

axioms

- Derived round-trip property: decrypt encrypted message

```
(defproperty decrypt-inverts-encrypt
  (m  :value (+ (random-natural) 2)
   n  :value (random-natural)
   xs :value (random-list-of (random-between 0 (- m 1)) :size (+ n 1))
      :where (and (natp m) (> m 1)
                  (consp xs) (true-listp xs) (code-listp m xs)))
  (equal (decrypt m (encrypt m xs)) xs))
```

Accommodations for ACL2

- ACL2 logic requires all functions to be total
  - Definition admitted to ACL2 logic only after proving termination
  - Functions must terminate for all inputs
- An implication for programming

```
(defun encrypt (m xs)
  (if (consp (cdr xs))
      (cons (encrypt-pair m (car xs) (cadr xs))
            (encrypt m (cdr xs)))
      (list (encrypt-pair m (car xs) (1- m)))))
```

  - Conventional definition of encrypt use (if (null (cdr xs)) …)
  - The Boolean (null xs) can fail to trigger termination for some inputs
    - ✓ For example, if xs is an atom other than nil
    - ✓ Using (consp xs), or (atom xs) or (endp xs), avoids this problem
    - ✓ Programmer expects all inputs to be true lists (nil-terminated), but ACL2 requires totality, so the definition must cover all cases
- Other tricks (among many)
  - Count down on naturals and use zp or posp for termination test
    - ✓ Suggests induction scheme that works (counting up is trickier)
  - Lemma needed when "rest" is not used for list-shortening recursions

Binary Numerals

```
(defun numb (x) ; number denoted by binary numeral x
  (if (consp x)
      (if (= (first x) 1)
          (+ 1 (* 2 (numb (rest x))))
          (* 2 (numb (rest x))))
      0))
(defun bits (n)              ; binary numeral for n
  (if (zp n)
      nil                                ; bits0
      (cons (mod n 2)                    ; bits1
            (bits (floor n 2)))))
```

axioms

- ❑ Derived round-trip property: number to numeral and back

```
(defproperty numb-inverts-bits
  (n  :value (random-natural))
  (= (numb (bits n)) n))
```

Arithmetic on Binary Numerals

```
(defun add-1 (x)
  (if (consp x)
      (if (= (first x) 1)
          (cons 0 (add-1 (rest x))) ; add11
          (cons 1 (rest x)))        ; add10
      (list 1)))
(defun add-c (c x)
  (if (= c 1)
      (add-1 x)                     ; addc1
      x))                           ; addc0
(defun add (c0 x y)
  (if (and (consp x) (consp y))
      (let* ((x0 (first x))
             (y0 (first y))
             (a  (full-adder c0 x0 y0))
             (s0 (first a))
             (c1 (second a)))
        (cons s0 (add c1 (rest x) (rest y)))) ; addxy
      (if (consp x)
          (add-c c0 x)                        ; addx0
          (add-c c0 y))))                     ; add0y
```

axioms

- Derived property: add numerals or add numbers

```
(defthm add-ok
  (= (numb (add c x y))
     (+ (numb (list c)) (numb x) (numb y))))
```

Multiplication, Too

```
(defun my1 (x y)                          ; x,y: binary numerals, y non-empty
  (if (consp x)
      (let* ((m  (my1 (rest x) y)))
        (if (= (first x) 1)
            (cons (first y) (add 0 (rest y) m)) ; mul1xy
            (cons 0 m)))                        ; mul0xy
      nil))                                     ; mul0y

(defun mul (x y)
  (if (consp y)
      (my1 x y)                                 ; mulxy
      nil))                                     ; mulx0
```

axioms

- Derived property: multiply numerals or multiply numbers

```
(defthm mul-ok
  (= (numb (mul x y))
     (* (numb x) (numb y))))
```

Another Accommodation for Proofs

- Avoid hypotheses in theorems when possible
  - "Normal" representation of binary numeral would be list of 0s and 1s
  - But, theorems would be implications with "list of 0s and 1s" hypothesis
- Avoiding the "list of 0s and 1s" hypothesis
  - Use 1 for 1-bit and anything else for 0 bit
  - Use consp (or atom or endp) to avoid requiring true lists
  - Most of the time, inputs will be lists of 0s and 1s, but mechanized logic is not constrained to those particular representations of numerals

```
(defun numb (x) ; number denoted by binary numeral x
  (if (consp x)
      (if (= (first x) 1)
          (+ 1 (* 2 (numb (rest x))))
          (* 2 (numb (rest x))))
      0))
```

Test for "1" or "not 1"

Note (Marco):

1. consp
2. 1, not 1 instead of 0, 1
3. Use dblchk until bugs seem to be fixed
4. Reading the acl2 report panel

Nested Recursion vs Tail Recursion

$f_0 = 0$
$f_1 = 1$
$f_{n+2} = f_{n+1} + f_n$

algebraic equations

```
(defun Fibonacci (n)
  (if (zp n)
      0
      (if (= n 1)
          1
          (+ (Fibonacci (- n 1))
             (Fibonacci (- n 2))))))
```

transcribed to ACL2 syntax
infeasible computation

```
(defun fib-tail (n a b)
  (if (zp n)
      a
      (fib-tail (- n 1) b (+ a b))))
(defun Fibonacci-fast (n)
  (fib-tail n 0 1))
```

tail-recursive version
O(n) computation

derived property

```
(defthm Fibonacci=Fibonacci-fast
  (implies (natp n)
           (= (Fibonacci n)
              (Fibonacci-fast n))))
```

lemmas for
mechanized proof

```
(defthm fib-tail-Fibonacci-recurrence-0
  (= (fib-tail 0 a b) a))
(defthm fib-tail-Fibonacci-recurrence-1
  (= (fib-tail 1 a b) b))
(defthm fib-tail-Fibonacci-recurrence
  (implies (and (natp n) (>= n 2))
           (= (fib-tail n a b)
              (+ (fib-tail (- n 1) a b)
                 (fib-tail (- n 2) a b)))))
```

ACL2 Sometimes Needs Hints

- Axiomatic properties of suffix function

```
(defun suffix (n xs)
  (if (posp n)
      (suffix (- n 1) (rest xs))  ; sfx1
      xs))                        ; sfx0
```

- Derived property: suffix reduces length

```
(defproperty suffix-reduces-length
  (xs :value (random-list-of (random-symbol))
   n  :value (random-natural)
      :where (and (consp xs) (posp n)))
  (<  (len (suffix n xs)) (len xs))
  :hints (("Goal" :induct (len xs))))
```

suggests induction strategy

- Theorem that Dracula sends to ACL2 logic

```
(defthm suffix-reduces-length
  (implies (and (consp xs) (posp n))
           (<  (len (suffix n xs)) (len xs))
  :hints (("Goal" :induct (len xs))))
```

Future Work

- Have: hundreds of defined properties
  - Ten years, three courses
  - Lectures
  - Homework projects
  - Exams
- Goal: web accessible archive
  - Notes and Dracula definitions for all properties
  - Lemmas and hints for ACL2 mechanized proof
- Target date: May 2012

The End