Logic and Computation – CS 2800
Fall 2019

Lecture 9

Property-based testing continued

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Property-based testing continued
  - `test?` and `thm`

Testing vs proving

- Testing:

```
(test? (implies (natp n)
                (equal (even-natp n)
                       (even-intp n))))
```

  - We are asking the tool to generate many examples and test on each example whether property holds
    - In this case, an example is one specific **`n`**
  - The test passes if the tool cannot find an example violating the property: a counter-example
  - More powerful than `check=` which tests just one example
- Theorem proving:

```
(thm (implies (natp n)
              (equal (even-natp n)
                     (even-intp n))))
```

  - We are asking the tool to prove that the property holds for every `n`
  - More powerful than `test?` – why?
  - Also theorem proving techniques fundamentally different from testing

Important: implicit for-all quantification in properties

- In the property below ***"for all n"*** is implied:

```
(implies (natp n)
         (equal (even-natp n)
                (even-intp n)))
```

- This really means *"for all n in the ACL2s universe"*, not just *"for all natural numbers n"* !

Summary: `thm` vs `test?`

- `test?` – possible outcomes:
  - — Counter-example: property does not hold (is false)
  - — All examples pass: property might or might not hold
  - — Proof: sometimes `test?` manages to actually prove the property => property holds (is true)

`test?` fails

`test?` passes

- `thm` – possible outcomes:
  - — Counter-example: property does not hold (is false)
  - — Unknown: `thm` could not prove it, did not find counter-example
  - — Proof: property holds (is true)

`thm` fails

`thm` passes

Structure of properties

- Usually our properties will be of the form

```
(implies H C)
```

hypothesis conclusion

- Usually H will be of the form

```
(and (R1 x1) (R2 x2) … (Rn xn) …)
```

  - Ri's are recognizers and xi's are variables appearing in C
  - The second … can be some extra assumptions
  - We must perform contract checking on all the non-recognizers in H
    - i.e., the … after the recognizers must satisfy its contracts, assuming prior assumptions hold
- C can be any Boolean expression
  - We must perform contract checking also for C
  - All functions in C must satisfy their contracts, assuming H holds

Examples

```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))
```

```
(definec even-intp (x: int) :bool
  (integerp (/ x 2)))
```

```
(test? (implies (natp n)
                (equal (even-natp n)
                       (even-intp n))))
```

Contract checking passes

```
(test? (implies (intp n)
                (equal (even-natp n)
                       (even-intp n))))
```

Contract checking fails. `(even-natp n)` requires n to be a nat. Result of `test?` untrustworthy!

Examples

```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))
```

```
(definec even-intp (x: int) :bool
  (integerp (/ x 2)))
```

```
(test? (implies (< 20/3 n)
                (equal (even-natp n)
                       (even-intp n))))
```

Contract checking fails.
`(< 20/3 n)`
requires n to be rational.

```
(test? (implies (and (natp n) (< 20/3 n))
                (equal (even-natp n)
                       (even-intp n))))
```

Contract checking passes.
A nat is also a rational.

Examples

```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))
```

```
(definec even-intp (x: int) :bool
  (integerp (/ x 2)))
```

This property holds:

```
(test?
 (implies
   (natp n)
   (equal (even-natp n)
          (even-intp n))))
```

What about this one?

```
(test?
 (implies
   (and (intp n) (< n 0))
   (equal (even-intp n)
          (even-natp (* n -1)))))
```

Alternative definition of `even-intp`:

```
(definec even-intp2 (x :int) :bool
  (if (natp x)
        (even-natp x)
        (even-natp (* x -1))))
```

Because of the two properties to the left, these two definitions are equivalent.

How would you express this equivalence in ACL2s?

Examples

```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))
```

```
(definec even-intp (x: int) :bool
  (integerp (/ x 2)))
```

This property holds:

```
(test?
 (implies
   (natp n)
   (equal (even-natp n)
          (even-intp n))))
```

What about this one?

```
(test?
 (implies
   (and (intp n) (< n 0))
   (equal (even-intp n)
          (even-natp (* n -1)))))
```

Alternative definition of `even-intp`:

```
(definec even-intp2 (x :int) :bool
  (if (natp x)
       (even-natp x)
       (even-natp (* x -1))))
```

Because of the two properties to the left, these two definitions are equivalent.

We can express this as an ACL2s theorem:

```
(thm (implies
          (intp x)
          (equal (even-intp x)
                 (even-intp2 x))))
```

Contract checking in `test?/thm` and in functions

```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))
```

```
(definec even-intp (x: int) :bool
  (integerp (/ x 2)))
```

Contract checking this

```
(test? (implies (and (natp n) (< 20/3 n))
                (equal (even-natp n)
                       (even-intp n))))
```

is the same as performing contract checking on this function:

```
(defunc test1 (n)
  :input-contract (and (natp n) (< 20/3 n))
  :output-contract (booleanp (test1 n))
  (equal (even-natp n)
         (even-intp n)))
```

In ACL2s the specification language is embedded into the programming language!

Quiz

- Consider the following statement:

```
(thm (equal (first x) (car x)))
```

A. It is indeed a theorem
B. It is not a theorem
C. It does not even pass contract checking

Quiz

- Consider the following incomplete statement:

```
(thm (implies (listp x)
              (equal (second x) ???)))
```

- What can we put in the place of ??? to make the statement a true theorem?

  A. `(car x)`
  B. `(cdr x)`
  C. `(car (cdr x))`
  D. `(cdr (car x))`
  E. Nothing. No matter what we put, it will not pass contract checking.

Demo

- See file 09-property-testing.lisp

Next

- Propositional logic