Property-Based Testing

Pete Manolios
Northeastern

Logic and Computation, 1/23/2019

Objectives

- Finish data definitions
- Controlling ACL2s
- Property-based testing
- test? & thm

Data Definitions

- Demo
- Singleton types
- Recognizers
- Enumerated Types
- Range Types
- Product Types
- Records
- Constructors & Accessors
- Listof Combinator
- Union Types
- Recursive Types
- Data-driven Function Definitions
- Mutually Recursive Data Types

Controlling ACL2s

- `:program` mode turns off theorem proving in ACL2s
  - no termination analysis is attempted
  - ACL2s will still *test* contracts and report any errors it finds
  - useful for prototyping & experimenting
- `:logic` mode is the default mode and allows you to switch back
  - you cannot define `:logic` mode functions if they depend on `:program` mode functions
- Other useful settings (we used some of them in HWK 2)
  - `(acl2s-defaults :set testing-enabled nil)`
  - `(set-defunc-termination-strictp nil)`
  - `(set-defunc-function-contract-strictp nil)`
  - `(set-defunc-body-contracts-strictp nil)`
- Documentation via `:doc`

Property-Based Testing

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

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

- Here is how we test properties in ACL2s

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

This is a property

- This gives us way more power than `check=` as ACL2s checks the property on a large number of examples (user-controlled); passes iff all checks pass

Theorem Proving

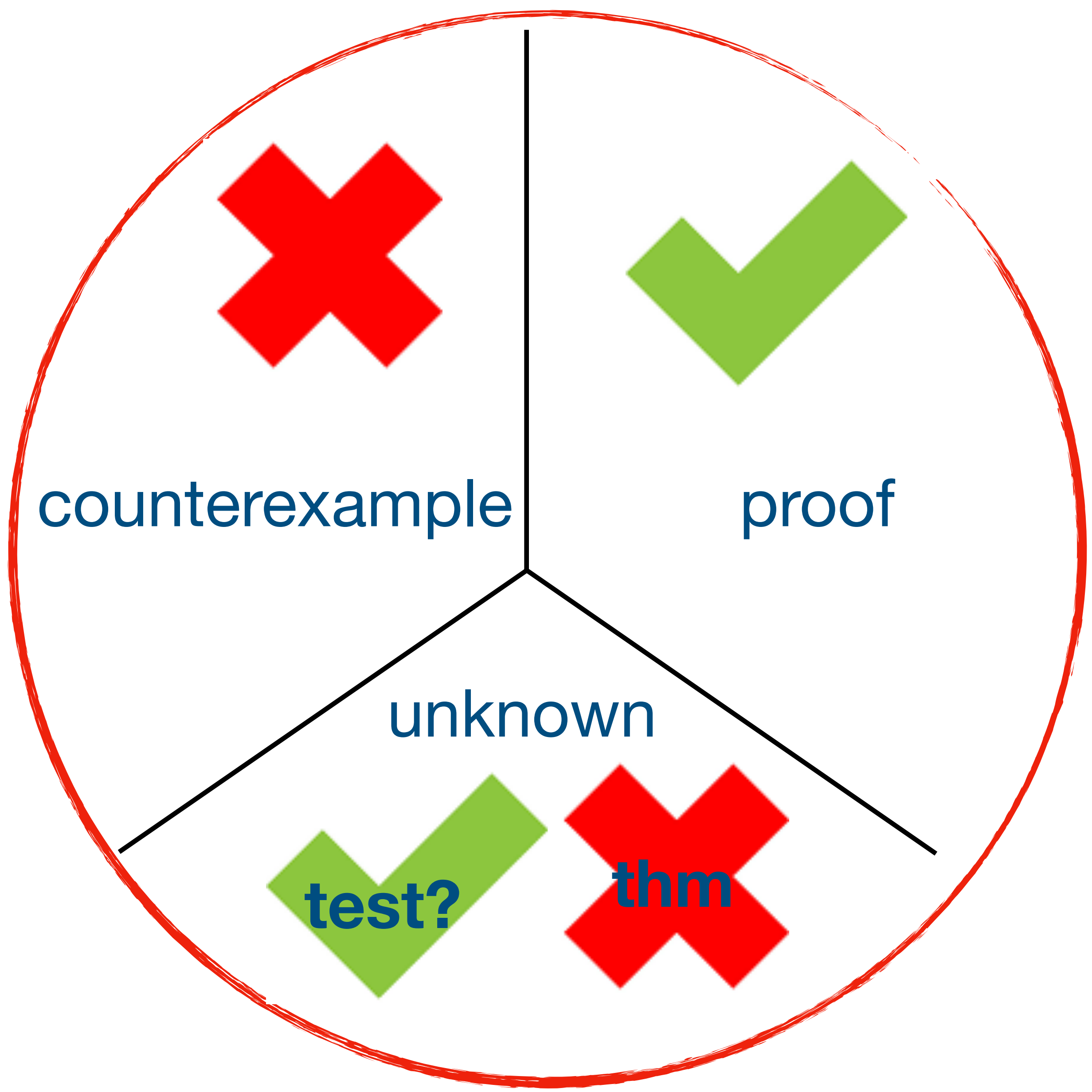
```
(definec even-natp (x :nat) :bool
  (natp (/ x 2)))

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

- Here is a way of proving theorems in ACL2s

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

- This gives us way more power than `check=` & `test?` as it corresponds to an *infinite* number of checks, i.e., it is always true; passes if ACL2s proves it

Thm vs Test? on Properties



test? vs thm

- test?
  - user only has to provide properties
  - ACL2s does the rest: it automatically looks for counterexamples
  - a kind of “light-weight” formal methods
- thm
  - highest assurance level
  - guarantees that the property is *always* true
  - requires human expert to interactively drive theorem prover
  - a kind of “heavy-weight” formal methods
- Practical considerations
  - start with test?’s and convert to thm’s based on budget/risk analysis

Structure of Properties

- Structure of `test?/thm` is `(test?/thm (implies H C))`
- Where `H` (Hypothesis) is of the form `(and (R1 x1)…(Rn xn)…)`
  - All the `Ri`s are recognizers & the `xi`s are variables in `C` (Conclusion)
  - The second ... can be some other, extra assumptions
- `C` is a boolean expression
- We must perform contract checking on all the non-recognizers in `H`
  - The stuff after the recognizers must satisfy its contracts, assuming everything before it holds
- We must perform contract checking 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
  (intp (/ x 2)))
```

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

Contract checking passes

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

Contract checking fails
`(even-natp n)`
requires n to be a *nat*

Examples

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

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

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

Contract checking passes

< knows n is a rational

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

Contract checking fails

< does not know that n is a rational

Examples

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

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

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

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

Does this property hold?

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

The two properties characterize `even-intp` in terms of `even-natp`, so they show another way we could have defined `even-intp`

Unifying Observation

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

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

Contract checking a `test?/thm` is equivalent to contract checking functions

For example, contract checking the test? is equivalent to checking the function

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

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

In ACL2s, the specification language and programming language are the same!

Demo

- Examples in the slides
- More examples

Next Time

- Property-Based Testing in Industry
- Fuzzing for Security Testing

- Next Week: Propositional Logic
- Read Chapter 3