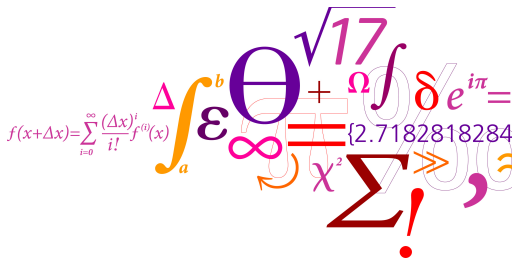


02257 Applied Functional Programming

Property-based testing: An appetizer

Michael R. Hansen



DTU Compute

Department of Applied Mathematics and Computer Science

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, Claessen and Hughes, 2000

- Random generation of values of arbitrary types
- Properties are expressed (programmed) as Boolean-valued functions

```
let rec sort xs = .....  
let rec ordered xs = ...  
  
// Test that: for all lists xs: ordered(sort xs)  
let sortProp (xs: int list) = ordered(sort xs)  
  
let _ = Check.Quick sortProp  
    Ok, passed 100 tests.
```

The `QuickCheck` tool has been ported to many languages. We look at `FsCheck` for the .Net platform

Consult <https://fscheck.github.io/FsCheck/> concerning installation and resources

Tests:

- easy to write
 - can reveal errors
 - show correctness of a very limited number of concrete cases
- low level of abstraction

Verification:

- complicated to complete
- provide guarantees
- focus of correctness properties

high level of abstraction

Tests: ...

Property-based testing

- focus on properties of programs enhances understanding
- construction of programs
- a randomly generated sample covers edge cases and typical situations
- Short counter-examples are useful
- gives high confidence
- limited effort

high level of abstraction

Verification: ...

Testing for correctness wrt. a reference model (I)

```
#r "nuget: FsCheck";;
open FsCheck

let rec sumA xs acc = match xs with
    | []      -> 0
    | x::xs   -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Quick sumRefProp;;
Falsifiable, after 2 tests (2 shrinks) (StdGen ..... :
Original:
[-2; -1]
Shrunk:
[1]
```

- uses built-in generators for lists
- tool provides a **short counterexample**

Testing for correctness wrt. a reference model (II)

```
let rec sumA xs acc = match xs with  
    | []      -> acc  
    | x::xs   -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;  
let _ = Check.Quick sumRefProp;;  
Ok, passed 100 tests.
```

- default is 100 random tests
- can be configured

Test cases are exposed using `Check.Verbose` as follows:

```
let sumRefProp xs = List.sum xs = sumA xs 0;;  
let _ = Check.Verbose sumRefProp;;
```

```
0:
```

```
[-2]
```

```
.....
```

```
99:
```

```
[-1; 0; -1; -1; 2; 1; -1; 0; 0; 5; -1; 1; -1; 0; 0; -1; 2;  
1; -1; 1; -1; 0; -1; -1; -1; -1; 1; -1; 1; 1; 1; 0; -2; 1;  
1; 0; -1; 0; -1; -1; -2; 2; 0; 1; -1; -1; 1; 1; 0; 0; -1; 0]  
Ok, passed 100 tests.
```

Correctness properties are Boolean-valued functions

A **property** is a Boolean-valued function with type

$$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \text{bool}$$

Append is associative:

```
let appendAssocProp xs ys zs =  
    xs @ (ys @ zs) = (xs @ ys) @ zs  
'a list -> 'a list -> 'a list -> bool when 'a : equality
```

The associative law can be tested on some examples:

```
let test = appendAssocProp [1;2] [3;4;5] [6;7;8;9];;
```

- But it requires discipline to come out with a suitable test suite

How can we get confidence in tests that should validate that

```
appendAssocProp xs ys zs
```

holds for all lists *xs*, *ys* and *zs*?

Property-based testing

Given

- property F with type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \text{bool}$
- with input variable x_1, \dots, x_n

the library FsCheck supports

- generation of random values for x_1, \dots, x_n
- test whether F holds for all sample values
- presentation of a short counter-example when F is falsified

where x_i can have any **monomorphic type**.

```
open FsCheck
```

```
let appendAssocProp (xs: int list) ys zs =  
    xs @ (ys @ zs) = (xs @ ys) @ zs;;
```

```
Check.Quick appendAssocProp;;  
Ok, passed 100 tests.  
val it : unit = ()
```

Properties/problems concerning floating point numbers

```
1.1 + 0.1 = 1.2;;  
val it : bool = false
```

```
1.0/0.0;;  
val it : float = infinity  
-1.0/0.0;;  
val it : float = -infinity
```

```
infinity + -infinity;;  
val it : float = nan
```

- Issues with approximations
- Issues with non-standard numbers

```
Check.Verbose (fun (x:float) (y:float) -> x+y = y+x);;  
Falsifiable, after 52 tests (1 shrink) (StdGen (167707349,  
Original:  
18.38287219  
nan  
Shrunk:  
0.0  
nan
```

A solution using NormalFloat and a tolerance

The type `NormalFloat` does not contain NaN, infinity and -infinity.

```
let within (tol:float) x y = abs (x-y) < tol

let assocTol tol x y z = within tol (x+(y+z)) ((x+y)+z);;

// For all NormalFloats xn yn zn:
//   x+(y+z) = (x+y)+z
//   within tolerance 1.0E-10

let assocTolNF xn yn zn =
  let x = NormalFloat.op_Explicit xn
  let y = NormalFloat.op_Explicit yn
  let z = NormalFloat.op_Explicit zn
  assocTol 1.0E-10 x y z;;

let _ = Check.Quick assocTolNF;;
// Ok, passed 100 tests.
```

Property-based testing supports testing at a high level of abstraction

- Focus is on fundamental properties – not on concrete test cases
- You write programs for properties – not concrete test cases
- Properties are tested automatically
- Short counter-examples are found — when properties are falsified

More features and techniques later