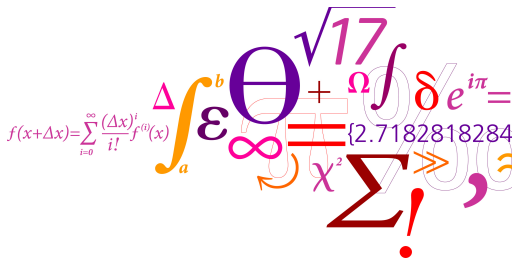


02257 Applied Functional Programming

Property-based testing: Properties, statistics and generators

Michael R. Hansen



DTU Compute

Department of Applied Mathematics and Computer Science

Property-based testing

- Statistical information about samples
- Making your own generators
- Tests involving preconditions

An example with a precondition

```
let rec ordered =  
  function  
  | [] | [_] -> true  
  | x::y::rest -> x<=y && ordered(y::rest);;  
  
let rec insert x =  
  function | [] -> [x]  
           | y::rest when x>y -> y::insert x rest  
           | ys -> x::ys;;  
  
let insertProp (x:int) xs =  
  not(ordered xs) || ordered(insert x xs);;  
  
Check.Quick insertProp;;  
Ok, passed 100 tests.
```

Is this a good test?

Classification of test cases (1)

The function `Prop.trivial` is used to count cases where a condition holds

Classification of trivial test cases:

```
let insertProp1 (x:int) xs =  
    (not (ordered xs) || ordered(insert x xs))  
    |> Prop.trivial (not (ordered xs));;
```

```
Check.Quick insertProp1;;  
// Ok, passed 100 tests (91% trivial).
```

- 91% of the generated lists `xs` are not ordered!!!

Classification of test cases (2)

The function `Prop.classify` is used to classify cases

A detailed classification example:

```
let insertClassify (x:int) xs =  
  (not (ordered xs) || ordered (insert x xs))  
  |> Prop.classify (not (ordered xs)) "false precondition"  
  |> Prop.trivial (List.length xs = 0)  
  |> Prop.classify (xs<>[] && ordered(x::xs)) "at-head"  
  |> Prop.classify (xs<>[] && ordered(xs@[x])) "at-tail"  
  |> Prop.classify (xs<>[] && ordered xs &&  
    not (ordered(xs@[x]) || ordered(x::xs))  
    "inside"
```

```
Check.Quick insertClassify;;  
// Ok, passed 100 tests.  
// 85% false precondition.  
// 7% at-head.  
// 4% trivial.  
// 4% at-tail.
```

- Is there a problem?

no insertion inside a list

The types `Arbitrary<'a>` and `Gen<'a>`

An `Arbitrary<'a>` instance comprises a

- a generator `g` of type `Gen<'a>` and
- a shrinker `f` of type `'a -> seq<'a>`

to be used when testing properties.

Good default implementation exists for most types.

A generator `g : Gen<'a>` can be considered

- a computation of a random value of type `'a`

A shrinker `f` is a function that, for a given counter example, returns a sequence of smaller/simpler values.

The module `Gen` can be considered a library for forming new generators

New generators can also be formed using F#'s computation expressions

`Gen<'a>` is a monad using Haskell terminology

Using functions from the modules `Arb` and `Prop`:

```
Arb.mapFilter: ('a -> 'a) -> ('a -> bool)
               -> Arbitrary<'a> -> Arbitrary<'a>

Prop.forAll: Arbitrary<'a> -> ('a -> 'Testable)
             -> Property

let orderedList =
  Arb.mapFilter List.sort ordered Arb.from<list<int>>

let insertWithArb x =
  Prop.forAll orderedList (fun xs -> ordered(insert x xs))

Check.Quick insertWithArb;;
// Ok, passed 100 tests.
```

```
let insertWithArbClassify x =  
  Prop.forAll  
    orderedList  
    (fun xs -> ordered(insert x xs)  
      |> Prop.classify ( ... ) "false precondition"  
      |> Prop.trivial (List.length xs = 0)  
      |> Prop.classify ( ... ) "at-head"  
      |> Prop.classify ( ... ) "at-tail"  
      |> Prop.classify ( ... ) "inside" );;
```

```
Check.Quick insertWithArbClassify;;  
(*  
  Ok, passed 100 tests.  
  45% at-head.  
  44% at-tail.  
  10% inside.  
  1% trivial. *)
```


A new generator for float

From the modules **Gen** and **Arb**

```
Gen.map: ('a -> 'b) -> Gen<'a> -> Gen<'b>  
Arb.generate<NormalFloat>: Gen<NormalFloat>
```

Thus **Gen.map** lifts a conversion function to a generator conversion.

```
let myFloatGen =  
  Gen.map NormalFloat.op_Explicit  
    Arb.generate<NormalFloat>;;  
// val myFloatGen: Gen<float>
```

Can be registered and used in testing:

```
type MyGenerators =  
  static member float() =  
    new Arbitrary<float>() with  
      override x.Generator = myFloatGen  
      override x.Shrinker f = seq [f / 2.0]
```

```
Arb.register<MyGenerators>()
```

```
Check.Quick (fun (x:float) y -> x+y = y+x);;  
//Ok, passed 100 tests.
```

```
Gen.constant: 'a -> Gen<'a>
```

```
Gen.oneof: seq<Gen<'a>> -> Gen<'a>
```

```
Gen.map2: ('a->'b->'c) -> Gen<'a> -> Gen<'b> -> Gen<'c>
```

```
Gen.sized: (int -> Gen<'a>) -> Gen<'a>
```

```
Gen.frequency: seq<int * Gen<'a>> -> Gen<'a>
```

- Self-explaining using names and types

A simple example using an recursive type:

```
type E = | X | C of int | Add of E*E;;
```

```
let rec eval x = function  
  | X -> x  
  | C n -> n  
  | Add(e1,e2) -> eval x e1 + eval x e2;;
```

```
let prop1 x e = eval x e = eval x (Add(e,C 0));;
```

- Let us make our own generator

An un-safe generator

A recursive declaration of a generator:

```
let leafGen = Gen.oneof [Gen.constant X;  
                        Gen.map C Arb.generate<int>]  
// val leafGen : Gen<E>  
  
let rec unSafeGen() =  
    Gen.oneof [leafGen;  
              Gen.map2 (fun x y -> Add(x,y))  
                    (unSafeGen())  
                    (unSafeGen())];;  
// val unSafeGen : unit -> Gen<E>
```

After registering the generator:

```
let _ = Check.Verbose prop1;;  
// Stack overflow.
```

We must control the size of the generated trees

An safe generator

A recursive declaration of a generator with size control:

```
let safeGen() =  
  let rec myE n =  
    match n with  
    | 0 -> leafGen  
    | _ -> let egen = myE (n/2)  
            Gen.map2 (fun x y -> Add(x,y)) egen egen  
  Gen.sized myE;;
```

- **egen** is a generator of a tree (not a specific tree)
- Termination is guaranteed due to the use of the size parameter

but

```
let egen = myE (n/2)  
Gen.oneof[leafGen;  
  Gen.map2 (fun x y -> Add(x,y)) egen egen]
```