CMSC 330: Organization of Programming Languages

Property-Based Random Testing

How do Test a Program?

- A code tester walks into a bar
  - Orders a beer
  - Orders ten beers
  - Orders 2.15 billion beers
  - Orders -1 beer
  - Orders a nothing
  - Orders a lizard
  - Tries to leave without paying

What is in the secret tests

- Run your code on Linux
- Run your code on Windows
- Run your code Mac
- Run your code on Android
- Run your code 1000 times
- Run your code on a 20-year old computer

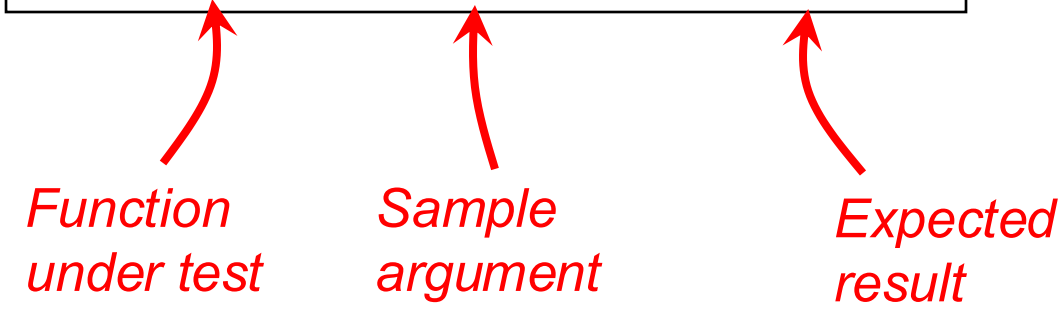
Let's test **rev** (list reverse) …

```
let rec rev l =
  match l with
    [] -> []
    | h::t -> rev t @ [h]
```

Let's test **rev** (list reverse) … with a unit test

```
let rec rev l =
  match l with
    [] -> []
    | h::t -> rev t @ [h]
```

```
let test_reverse =
   reverse [1;2;3] = [3;2;1]
```



Unit Testing

- Hard Coded Tests
- Difficult to write good unit tests
- Time Consuming
- Have to write many tests
- Repeated (redundant) Tests

Properties

- Instead of unit tests on *specific* inputs and outputs, what if we could test properties that hold for *all* inputs ?

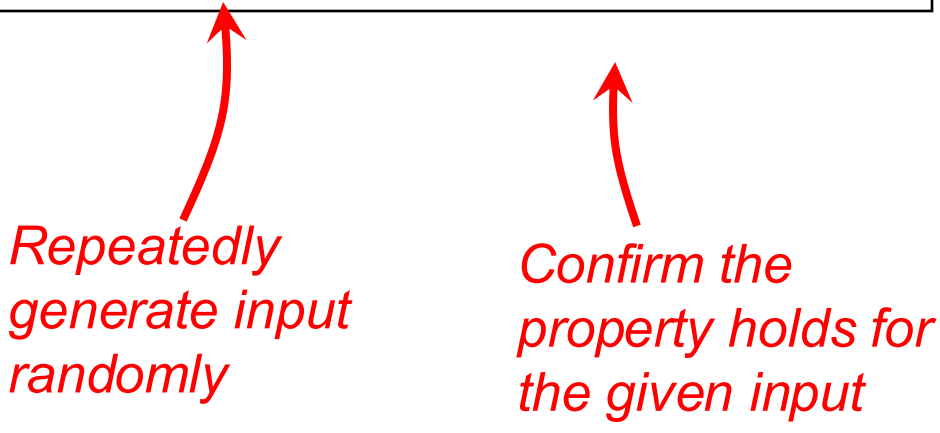
```
let prop_reverse l = rev (rev l) = l
```

  - I.e., reversing a list twice gives back the original list
- In other words, each of the following evaluates to **`true`**
  - **`prop_reverse []`**
  - **`prop_reverse [1; 2; 3]`**
  - **`prop_reverse [1.0; 2.22]`**

Property-based Testing

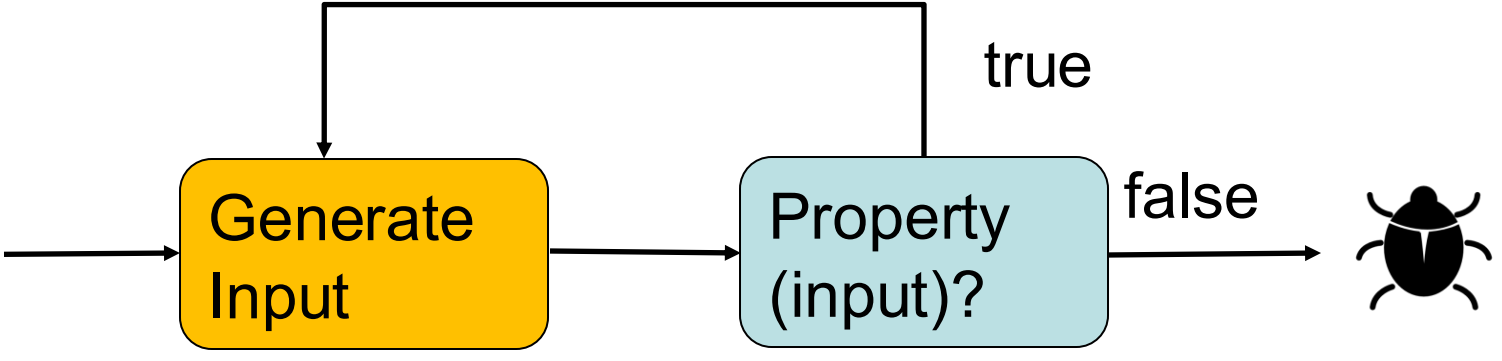
- a framework that repeatedly generates random inputs, and uses them to confirm that properties hold

```
let prop_reverse l =  rev (rev l) = l
```



QCheck: Property-Based Testing for OCaml

- QCheck tests are described by
  - A generator: generates random input
  - A property: `bool`-valued function



Setting Up QCheck

- Install

```
opam install qcheck
```

- Open the Qcheck module

```
open QCheck
```

- in utop, before `open QCheck`

```
#require "qcheck"
```

- In dune file

```
(libraries qcheck)
```

Let's Test Our Property

```
let prop_reverse l = rev (rev l) = l
```

```
open QCheck;;

let test =
    Test.make
   ~count:1000
   ~name:"reverse_test"
   (list small_int)
  (fun x-> prop_reverse x)
```

Test 1000 times

`:int list arbitrary`

Generates a random int list

...and tests the property

Let's test *properties* of **reverse**…

```
let prop_reverse l = rev (rev l) = l
```

```
open QCheck;;
let test = Test.make ~count:1000 ~name:"reverse_test"
(list small_int) (fun x-> prop_reverse x);;
```

- Run the test

```
QCheck_runner.run_tests ~verbose:true [test];;
```

```
generated  error  fail  pass / total     time test name
[✓] 1000     0     0    1000 / 1000     0.2s reverse_test
============================================================
success (ran 1 tests)
```

Test 1000 times

Buggy Reverse

```
let rev l = l  (* returns the same list *)
```

The property did not catch the bug!

```
let prop_reverse l = rev (rev l) = l
```

A simple unit test would catch the bug

```
let test_reverse = rev [1;2;3] = [3;2;1]
```

Another Property

```
let prop_reverse2 l1 m l2 =
   rev (l1 @ [m] @ l2) = rev l2 @ [m] @ rev l1
```

```
rev [1;2]@[3]@[4;5] = rev [4;5] @ rev [3] @ rev [1;2]
```

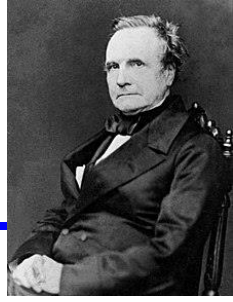
```
let test = QCheck.Test.make ~count:1000
 ~name:"reverse_test2"
 (triple (list small_int) small_int (list small_int))
 (fun(l1,m,l2)-> prop_reverse2 l1 m l2)
```

```
:(int list * int * int list) arbitrary
```

Generates `l1,x,l2`

```
QCheck_runner.run_tests [test];;
success (ran 1 tests)
- : int = 0
```

Lesson learned: Garbage in Garbage out



On two occasions I have been asked, –"*Pray, Mr. Babbage, if you put into the machine wrongfigures, will the right answers come out?*" In one case a member of the Upper, and in the other a member of the Lower, House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

– Charles Babbage, 1864

Bad generators and properties produce bad results.

Another example: Let's test **delete**…

```
let rec delete x l = match l with
    [] -> []
  | (y::ys) -> if x = y then ys
               else y::(delete x ys)
```

```
let prop_delete x l =
   not (List.mem x (delete x l))
```

x *should not be a member if deleted.*

Testing **delete**…

```
let prop_delete x l =
    not (List.mem x (delete x l))


let test = Test.make ~count:1000
~name:"delete_test"
(pair small_int (list small_int))
(fun(x,l)-> prop_delete  x l)


QCheck_runner.run_tests [test];;
```

Generate an `int` and an `int list`

Let's test *properties* of **delete**…

```
--- Failure ----------------------------------------

Test reverse_test failed (11 shrink steps):
(0, [0; 0])
==================================================
failure (1 tests failed, 0 tests errored, ran 1 tests)
- : int = 1
```

Delete only deleted the first occurrence

No recursive call!

```
let rec delete x l = match l with
    [] -> []
  | (y::ys) -> if x = y then ys
               else y::(delete x ys)
```

`delete 2 [2;2;3]  returns [2;3]`

Property: is_sorted

- Whether a list is sorted in non-decreasing order

```
let rec is_sorted lst =
 match lst with
  | [] -> true
  | [h] -> true
  | h1::(h2::t as t2) -> h1 <= h2 && is_sorted t2
```

`Arbitrary` Handles Random Inputs

- An `'a arbitrary` represents an "arbitrary" value of type `'a`
- It is used to describe how to
  - generate random values
  - shrink them (make counter-examples as small as possible)
  - print them

- `small_int:` `int arbitrary`
- `list:` `'a arbitrary -> 'a list arbitrary`
- `triple:`

```
'a arbitrary ->
  'b arbitrary ->
  'c arbitrary -> ('a * 'b * 'c) arbitrary
```

Arbitrary: The Details

```
type 'a arbitrary = {
   gen: 'a Gen.t;
   print: ('a -> string) option; (** print values *)
   small: ('a -> int) option; (** size of example *)
   shrink: 'a Shrink.t option; (** shrink to smaller examples *)
   collect: ('a -> string) option; (** map value to tag, and group by tag *)
   stats : 'a stat list;  (** statistics to collect and print *)
}
```

Build an Arbitrary

```
make :
   ?print:'a Print.t ->
   ?small:('a -> int) ->
   ?shrink:'a Shrink.t ->
   ?collect:('a -> string) ->
   ?stats:'a stat list -> 'a Gen.t -> 'a arbitrary
```

- Build an arbitrary that generates random ints

```
# make (Gen.int);;
- : int arbitrary =
{gen = <fun>; print = None; small = None; shrink = None;
    collect = None;stats = []}
```

Random Generator

- `'a QCheck.Gen.t` is a function that takes in a Pseudorandom number generator, uses it to produce a random value of type `'a`.
- For example, `QCheck.Gen.int` generates random integers, while `QCheck.Gen.string` generates random strings. Let us look at a few more of them:

```
module Gen :
    sig
      val int : int t
      val small_int : int t
      val int_range : int -> int -> int t
      val list : 'a t -> 'a list t
      val string : ?gen:char t -> string t
      val small_string : ?gen:char t -> string t
      ...
    end
```

Sampling Generators

```
Gen.generate1 Gen.small_int
7

Gen.generate ~n:10 Gen.small_int
  int list =[6;8;78;87;9;9;6;2;3;27]
```

Sampling Generators

- Generate 5 int lists

```
let t = Gen.generate ~n:5 (Gen.list Gen.small_int);;
val t : int list list =[[4;2;7;8;…];…;[0;2;97]]
```

- Generate two string lists

```
let s = Gen.generate ~n:2 (Gen.list Gen.string);;
val s : string list list =[[ "A";"B";…]; ["C";"d";…]]
```

Combining Generators

```
frequency:(int * 'a) list ->'a 'a Gen.t
```

- Generate 80% letters, and 20% space

```
Gen.generate ~n:10
      (Gen.frequency [(1,Gen.return ' ');
      (3,Gen.char_range 'a' 'z')]);;

- : char list=['i';' ';'j';'h';'t';' ';' ';' ';'k';'b']
```

Shrinking

- Our Delete example without shrinking…

```
--- Failure ------------------------------
-
Test anon_test_1 failed (0 shrink steps):

(7, [0; 4; 3; 7; 0; 2; 7; 1; 1; 2])
```

- …and with:

Where's the bug?

```
--- Failure ------------------------------
-
Test anon_test_1 failed (8 shrink steps):

(2, [2; 2])
```

Shrinking

How do we go from this…

```
(7, [0; 4; 3; 7; 0; 2; 7; 1; 1; 2])
```

…to this?

```
(2, [2; 2])
```

List of "smaller" inputs

- Given a *shrinking function* `f : : 'a -> 'a list`
- And a counterexample `x : : 'a`
- Try all elements of `(f x)` to find another failing input…
- Repeat until a minimal one is found.

Shrinkers

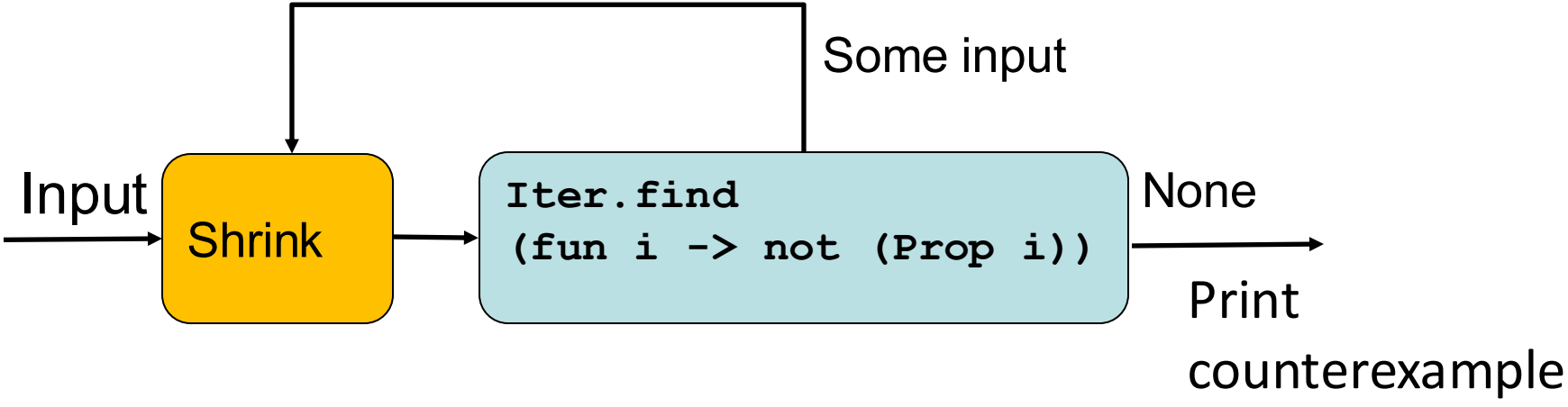
- A shrinker attempts to cut a counterexample down to something more comprehensible for humans

- A QCheck shrinker is a function from a counterexample to an iterator of simpler values:

```
'a Shrink.t = 'a -> 'a QCheck.Iter.t
```

Shrinkers and iterators in QCheck

- Given a counterexample, QCheck calls the iterator to find a simpler value, that is still a counterexample



After a successful shrink, the shrinker is called again.

Shrinkers

QCheck's **`Shrink`** contains a number of builtin shrinkers:

- **`Shrink.nil`** performs no shrinking
- **`Shrink.int`** for reducing integers
- **`Shrink.char`** for reducing characters
- **`Shrink.string`** for reducing strings
- **`Shrink.list`** for reducing lists
- **`Shrink.pair`** for reducing pairs
- **`Shrink.triple`** for reducing triples

Printers

- Type of printers
  - `type 'a printer = 'a -> string`
- Printers for primitives:
  - `val pr_bool : bool printer`
  - `val pr_int  : int  printer`
  - `val pr_list : 'a printer ->`
  - `                'a list printer`

Summary

- We've taken a brief look at QCheck Property Based Testing
  - how to generate random tests
  - how to build an arbitrary
  - how to use shrinkers