CMSC 330: Organization of Programming Languages

Property-Based Random Testing
Testing is Hard

• This happened in CMSC330 final exam
• Question: write a function sort ('a list -> 'a list) that receives an int list and returns a sorted list

• Student Answer:

```ocaml
let sort lst = [1;2;3]
```
Testing is Hard

• Question: write a function sort ('a list -> 'a list) that receives an int list and returns a sorted list

• Student Answer:

\[
\text{let sort lst } = [1;2;3] ;;
\]

(* this indeed returns a sorted list. This student received full credit for the question*)
Testing is Hard

- Question: write a function sort ('a list -> 'a list) that receives an int list the returns sorted list

Changed to:

Question: write a function sort ('a list -> 'a list) that receives an int list, sorts the list in non-descending order, and returns this sorted list.

Also:
1. Returned list must be a permutation of the input. Permutation is defined as ……
2. You can add recursive helper functions
3. You can use fold and map
Testing is Hard

By the time you finish reading the instructions, exam time is up.
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
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

• NO. We don’t do that
Let’s test `reverse`…

Not tail recursive

```
let rec reverse l =
    match l with
    [] -> []
    | h::t -> reverse t @ h
```
Let’s test `reverse`…

Unit tests…

<table>
<thead>
<tr>
<th>Function under test</th>
<th>Sample argument</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>test_reverse</code></td>
<td><code>reverse [1;2;3]</code></td>
<td><code>[3;2;1]</code></td>
</tr>
</tbody>
</table>
Unit Testing

• Hard Coded Tests
• Difficult to write good unit tests
• Time Consuming
• Have to Write many tests
• Repeated Tests
Properties

- Instead of hard coded unit tests, we should test the properties.

- Determine whether an integer is even

\[
\text{let is\_even } n = n \mod 2 = 0
\]
QCheck: Property-Based Testing for OCaml,

- QCheck tests are described by
  - A generator: generates random input
  - A Property: Boolean valued function
Let’s test properties of `reverse`…

Write a property that should hold for all inputs:

Random arguments

```
let prop_reverse l =
    reverse (reverse l) = l
```

Reverse of the reversed list is itself
Let's test properties of reverse...

```
let prop_reverse l = reverse (reverse l) = l
```

open QCheck;;

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

Generate an integer and a list

...and test the property
Let’s test properties of reverse…

```ocaml
let prop_reverse l = reverse (reverse l) = l

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

Run the test

```ocaml
QCheck.Test.check_exn test;;
- : unit = ()
```
Buggy Reverse

let reverse l = l (* returns the same list *)

The property did not catch the bug!

let prop_reverse l = reverse (reverse l) = l

A simple unit test would catch the bug

let test_reverse = reverse [1;2;3] = [3;2;1]
Reverse Property another take

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

rev [1;2]@[3]@[4;5] = rev [4;5] @ [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,x,l2)-> prop_reverse2 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 wrong figures, 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)

Unit Test

let test_delete =
  delete 2 [1;2;3] = [1;3]
Let's test properties of delete…

Write a property that should hold for all inputs:

Random arguments

let prop_delete x l =
not (member x (delete x l))

x should not be a member of the result
Let’s test properties of delete…

let prop_delete x l =
    not (member x (delete x l))

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

QCheck_runner.run_tests [test];;
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
Let’s test properties of `delete`...

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

```ocaml
let prop_delete x l = not (member x (delete x l))
```

Note: No recursive call!
Properties: is_sorted

- Whether a list is sorted in non-decreasing order

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

Generator
- Produces random data to test the property

Shrinker
- Minimizes counterexamples

Printer
Generators

- Abstract type of generators:
  - `type 'a gen`
- Sampling generators:
  - `val generate : 'a gen -> 'a`

```ocaml
> 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]
```
Generators

Generate 5 int lists

let t = Gen.generate ~n:5 (Gen.list Gen.small_int);;

t : int list list =[[4;2;7;8;...];...;[0;2;97]]

Get the length of each list:

List.map (fun x -> List.length x) t;;

Generate two string lists

let s = Gen.generate ~n:2 (Gen.list Gen.string);;
Generators

- Composite generators:
  
  ```
  val always : 'a -> 'a arbitrary
  ```

- Composite generators:
  
  ```
  val pair : 'a arbitrary -> 'b arbitrary -> ('a * 'b) arbitrary
  ```
Generators Examples

(* Always generate 42 *)

```ml
generate1 (QCheck.always 42) 42
```

(* generate a (int * bool) pair list *)

```ml
generate1 (Gen.list ((pair small_int bool).gen));
```

```
[(4,true); (0,false); (7, false)]
```
Generators

- Combining generators:
  ```
  val frequenc:(int * 'a) list ->'a 'a arbitrary
  ```

**Generate 80% small int and 20% int**

```plaintext
Gen.generate ~n:10
(frequency [(1,int);(4,small_int)]).gen;;
```

- : int list =
  ```plaintext
  [3; 4; -1745206713219709656; 9; 8; -4194515886393930669; 78; 1; 7; 35]
  ```
Generators

- Combining generators:
  
  ```
  val frequency : (int * 'a) list -> 'a 'a arbitrary
  
  Generate 75% 'a' and 25% 'b'
  
  let g = (frequencyl [(3,'a');(1,'b')]).gen;;
  Gen.generate ~n:8 g;;
  - : char list =
  [ 'b'; 'a'; 'a'; 'b'; 'b'; 'a'; 'a'; 'a' ]
  ```
Shrinking

- Our example without shrinking…

```plaintext
--- Failure -----------------------------------------
Test anon_test_1 failed (0 shrink steps):
(7, [0; 4; 3; 7; 0; 2; 1; 1; 2])
```

- ...and with:  

```plaintext
--- Failure -----------------------------------------
Test anon_test_1 failed (8 shrink steps):
(2, [2; 2])
```

Where's the bug?
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 \( f :: \text{`a -> `a list} \)
• And a counterexample \( x :: \text{`a} \)
• 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
Arbitraries – *Putting it all together*

- Represents an "arbitrary" value of type
- Combination type
  - `type 'a arbitrary`
- Combines all three components
  - Printer
  - Shrinker
  - Generator
An arbitrary integer:

```make
make Gen.int
- : int arbitrary =
```
Case Study: Binary Search Trees

type tree =
  | Leaf
  | Node of int * int * tree * tree

val nil :: tree
val insert :: int -> int -> tree -> tree
val delete :: int -> tree -> tree
val find :: int -> tree -> int option
Val valid :: tree -> bool
Binary Search Trees - Generation

type tree =
  | Leaf
  | Node of int * int * tree * tree

let rec insert (x,y) t =
  match t with
  | Leaf -> Node (x,y, Leaf, Leaf)
  | Node (k,v, l, r) ->
    if x = k then Node (k,y,l,r)
    else if x < k then Node (k,v, insert (x,y) l, r)
    else Node (k,v, l, insert (x,y) r)
Binary Search Trees - Generation

```ocaml
let tree_gen m =
  match n with
  | 0 -> Leaf
  | m -> let lst =
      Gen.generate ~n:m (Gen.pair Gen.nat Gen.nat) in
      List.fold_left (fun a (k,v) ->
        insert (k,v) a)
      Leaf l
```
let rec print_tree = function
| Leaf -> "Leaf"
| Node (k,v,l,r) ->
    "Node (" ^ (string_of_int k) ^ " ,")
    ^ (string_of_int v) ^ ","
    ^ (print_tree l) ^ ","
    ^ (print_tree r)
Validity Testing

- Test whether operations preserve invariant
  - let prop_insert_valid k v t =
    - valid (insert k v t)
  - let prop_delete_valid k t =
    - valid (delete k t)

- Test whether generation produces valid trees
  - let prop_gen_valid t =
    - valid t
Postcondition Testing

- What is the postcondition of find?
  - After calling find…
    - If the key is present, the result should be a Some
    - If the key is absent, the result should be None

By construction!

let prop_find_post_present k v t = find k (insert k v t) == Some v
let prop_find_post_absent k t = find k (delete k t) == None
Metamorphic Testing

- How does changing the input of insert change the result?

\[ O(n^2) \]

ideas
Metamorphic Testing

- How does changing the *input* of `insert` change the result?

  
  ```
  let prop_insert_insert (k,v) (k′,v) t =
  insert k v (insert k′ v′ t)
  ==
  insert k′ v′ (insert k v t)
  ```

  Is this really true?

--- Failure ----------------------------------

Test anon_test_1 failed (5 shrink steps):

```
((0,0), (0,1), Leaf)
```

Last insertion wins!
Metamorphic Testing

- How does changing the input of insert change the result?

```haskell
let prop_insert_insert (k,v) (k’,v) t =
    insert k v (insert k’ v’ t)
==
    if k == k’ then insert k v t else
    insert k’ v’ (insert k v t)
```

--- Failure -------------------------------------
- Test anon_test_1 failed (5 shrink steps):

```haskell
((0,0), (1,0), Leaf)
```

Order matters!
Metamorphic Testing

- How does changing the *input* of insert change the result?

  ```ml
  let bst_equiv t1 t2 =
  toList t1 == toList t2
  
  let prop_insert_insert (k,v) (k’,v’) t =
  bst_equiv
  (insert k v (insert k’ v’ t))
  (if k == k’ then insert k v t
   else insert k’ v’ (insert k v t))
  ```