OCaml Property-Based Testing

CSC 310 - Programming Languages

Unit Testing Example

- Code

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

- Unit test

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

Unit Testing

- Hard-coded test cases
- Difficult to write good unit tests
- Time consuming
- Need to write many tests
- Repeated tests

Properties

- Instead of unit tests on specific inputs and outputs, use properties that hold for all inputs
- Example: reversing a list twice is equal to the original list

```
let prop_reverse lst =
  rev (rev lst) = lst
```

Property-Based Testing

- A testing framework that repeatedly generates random inputs and uses them to confirm that a property hold
- Popularized by the Haskell QuickCheck library

QCheck

- QCheck is a property-based testing library for OCaml
- QCheck tests are described by:
  - A generator that generates random input
  - A property which is a Boolean valued function

Setting up QCheck

- Install

  ```
  opam install qcheck
  ```

- Open the `QCheck` module

  ```
  open QCheck
  ```

- In the top-level before `open QCheck`

  ```
  #require "qcheck"
  ```

QCheck Example

```
let prop_reverse lst = rev (rev lst) = lst;;

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

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

Buggy Reverse

- Buggy implementation

```
let rev lst = lst
```

- The property-based test does not catch the bug, but a simple unit test would

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

Add Another Property

```
let prop_reverse2 lst1 x lst2 =
  rev (lst1 @ [x] @ lst2) = rev lst2 @ [x] @ rev lst1

let test =
  QCheck.Test.make
  ~count:1000
  ~name:"reverse_test2"
  (triple (list small_int) small_int (list small_int))
  (fun (lst1,x,lst2) -> prop_reverse lst1 x lst2)

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

Another Example: `delete`

- Implementation

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

- Test

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

Testing `delete`

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

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

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

- Bug: `delete` only deletes the first occurrence

Property: `is_sorted`

- Check if 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`
- Used to describe how to:
  - generate random values
  - shrink them (make counter-examples as small as possible)
  - print them
- Examples

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

Arbitrary Details

```
type 'a arbitrary = {
  gen: 'a Gen.t;
  print: ('a -> string) option;
  small: ('a -> int) option;          (* example size *)
  shrink: 'a Shrink.t option;         (* shrink to smaller exam
  collect: ('a -> string) option; (* map value to tag, grou
  stats: 'a stat ist;                 (* statistics to collect
}
```

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


make (Gen.int);; (* generate random ints *)
```

Random Generator

- `'a QCheck.Gen.t is a function that takes in a pseudorandom number generator and uses it to produce a value of type'a‘`
- For example, `QCheck.Gen.int` generates random integers, while `QCheck.Gen.string` generates random strings
- `Gen` module:

```
module Gen :
  sig
    val int : int t
    val small_int : int t
    val list_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

Gen.generate ~n:10 Gen.small_int

Gen.generate ~n:5 (Gen.list Gen.small_int)

Gen.generate ~n:2 (Gen.list Gen.string)
```

Combining Generators

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

- Generate 75% letters and 25% spaces

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

Shrinking

Example testing delete; How do we do from this

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

to this?

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

- 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
  ```

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

- Printer type:

```
type 'a printer = 'a -> string
```

- Printers for primitives:
  - ‘pr_bool : bool printer
  - `pr_int : int printer`
  - `pr_list : 'a printer -> 'a list printer`

Summary

- QCheck Property Based library
  - how generate random tests
  - how to build an arbitrary
  - how to use shrinkers