SM2-TES: Functional Programming and Property-Based Testing, Day 4

Jan Midtgaard

MMMI, SDU

SDU

Last lecture's exercises

Lessons learned

- Many errors can lurk in a first specification
- You understand code better by exploring the spec.

  – mostly if it fails
- Sometimes the error is in the code,

  – and sometimes the error is in the spec.

Lessons learned

- Many errors can lurk in a first specification
- You understand code better by exploring the spec.
  – mostly if it fails
- Sometimes the error is in the code,
  – and sometimes the error is in the spec.
- To improve a spec., take the devil's advocate position:
  How could a wrong impl. pass this property?
- If an error is not caught by QuickCheck
  - then ask yourself: how could QuickCheck have found it?
  - (how) should I adjust my generator?
  - (how) should I adjust my property?

Outline

OCaml's module system

Shrinking

OCaml, recap

We've covered the core of OCaml:

- let-binding,
- pattern matching,
- lists and tuples,
- algebraic data types,
- recursive functions,
- references,
- records,
- exceptions,
- labeled and optional arguments,

QuickCheck, recap

. . . and studied the basics of QuickCheck along the way:

- properties
- generators (type-directed)
- statistics (observing generator distributions)
- . . .

and worked with these concepts in the `QCheck` framework

OCaml's module system

OCaml's module system (1/3)

OCaml code is typically split into modules

A module can contain both types, bindings, and other modules

We can address a binding or a type definition `f` in a module `M` with the familiar dot-syntax: `M.f`

For example:

- `String.length` refers to the `length` function bound inside the `String` module
- `Test.make` refers to the `make` function bound inside `QCheck`'s `Test` module
- `Gen.t` (the type of pure generators) refers to the `t` type defined inside `QCheck`'s `Gen` module

OCaml's module system (2/3)

One can open a module with **`open`** `M` to make `M`'s content immediately visible (without the `M.` prefix)

For example, we **`open`** `QCheck` to make the `QCheck` bindings, types, and sub-modules visible.

We thereby avoid having to qualify everything (`QCheck.Test.make`, `QCheck.Gen.t`, `...`)

OCaml's module system (2/3)

One can open a module with **`open`** `M` to make `M`'s content immediately visible (without the `M.` prefix)

For example, we **`open`** `QCheck` to make the `QCheck` bindings, types, and sub-modules visible.

We thereby avoid having to qualify everything (`QCheck.Test.make`, `QCheck.Gen.t`, `...`)

One can also open a module locally within the limited scope of `e` with **`let open`** `M` **`in`** `e` or `M.(e)`

OCaml's module system (2/3)

One can open a module with **`open`** `M` to make `M`'s content immediately visible (without the `M.` prefix)

For example, we **`open`** `QCheck` to make the `QCheck` bindings, types, and sub-modules visible.

We thereby avoid having to qualify everything (`QCheck.Test.make`, `QCheck.Gen.t`, ...)

One can also open a module locally within the limited scope of `e` with **`let open`** `M` **`in`** `e` or `M.(e)`

Alternatively one can simply abbreviate a module name. After **`module`** `QR = QCheck_runner` we can, e.g., refer to `QR.run_tests`

OCaml's module system (3/3)

OCaml modules can be further organized with

- signatures (think: interface) and
- functors (think: **`module -> module`** functions)

Example:

```
module Intset =
  Set.Make (struct
              type t = ...    (* element type *)
              let compare = ...
                        (* element comparison *)

            end)
```

OCaml's module system (3/3)

OCaml modules can be further organized with

- signatures (think: interface) and
- functors (think: **`module -> module`** functions)

Example:

```
module Intset =
  Set.Make (struct
              type t = int
              let compare n1 n2 =
                if n1 = n2 then 0 else
                  if n1 > n2 then 1 else -1
            end)
```

OCaml's module system (3/3)

OCaml modules can be further organized with

- signatures (think: interface) and
- functors (think: **`module -> module`** functions)

Example:

```
module Intset =
  Set.Make (struct
              type t = int
              let compare n1 n2 =
                if n1 = n2 then 0 else
                  if n1 > n2 then 1 else -1
            end)
```

Builtin maps are similar:

```
module Mymap = Map.Make(struct ... end)
```

OCaml modules and separate compilation (1/2)

We typically separate a module's implementation and interface into two separate files `x.ml` and `x.mli`.

This has the implicit effect of declaring a module named `X` and shadowing everything not listed in `x.mli`.

We can achieve the same effect by:

```
module X: sig
            (* contents of file x.mli *)
          end
        = struct
            (* contents of file x.ml *)
          end
```

Catch: Files are lower-case, but their module names are capitalized. Hence, the module in file `set.ml` is referred to as `Set`.

OCaml modules and separate compilation (2/2)

An example:

(In file `x.ml`:)

```
type t = string * (int -> string)
let a = ("foo",string_of_int)
let b = ("bar",fun _ -> "hello")
```

(In file `x.mli`:)

```
type t = string * (int -> string)

val b : t
```

Only `X.t` and `X.b` will be accessible from the outside.

OCaml modules and separate compilation (2/2)

An example:

(In file `x.ml`:)

```
type t = string * (int -> string)
let a = ("foo",string_of_int)
let b = ("bar",fun _ -> "hello")
```

(In file `x.mli`:)

```
type t

val b : t
```

Only `X.t` and `X.b` will be accessible from the outside.

Without a visible type definition, outsiders can never produce a value of type `X.t` other than `X.b`.

OCaml modules and separate compilation (2/2)

An example:

(In file `x.ml`:)

```
type t = string * (int -> string)
let a = ("foo",string_of_int)
let b = ("bar",fun _ -> "hello")
```

(In file `x.mli`:)

```
type t

val b : t
```

Only `X.t` and `X.b` will be accessible from the outside.

Without a visible type definition, outsiders can never produce a value of type `X.t` other than `X.b`.

"Filenames as modules" can be a bit confusing.

If we write

```
module S = struct let f = ... end
```

in a file `foo.ml` then from the outside we (need to) refer to `f` as `Foo.S.f`

Shrinking

Small and big counterexamples (1/2)

Suppose we try to QuickCheck a false claim, e.g., a buggy version of `rev_twice_test` from earlier:

```
let rev_twice_test =
  Test.make ~name:"rev␣twice"
    (list int)
    (fun xs -> List.rev (List.rev (List.rev xs)) = xs)
```

QCheck has post-processed the counterexample behind the scenes to present something easily understandable:

```
generated error fail pass / total     time test name
[✗]     1     0    1     0 /  100     0.0s rev twice

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

Test rev twice failed (132 shrink steps):

[0; -1]
```

Small and big counterexamples (2/2)

Suppose we try to QuickCheck the same claim, but disable the post-processing:

```
let rev_twice_test =
  Test.make ~name:"rev␣twice"
    (set_shrink Shrink.nil (list int))
    (fun xs -> List.rev (List.rev (List.rev xs)) = xs)
```

QCheck will present something less easily understandable (size varies with randomization seed):

```
generated error fail pass / total     time test name
[✗]     1     0    1     0 /  100     0.0s rev twice

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

Test rev twice failed (0 shrink steps):

 [4292509720569850 98; -2198275636720469779;
  -3640332438131865440; 3187320857408505647;
  2139558116497494025; -2370337062264846807;
  -3861516822461564113]
```

Big counterexamples

Once we move to user-defined data types the need becomes even more clear. Consider the false claim $\forall e.\, \texttt{interpret}\ (e + e) = \texttt{interpret}\ e$:

```
Test.make ~name:"test wrong"
  arb_tree
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

Big counterexamples

Once we move to user-defined data types the need becomes even more clear. Consider the false claim $\forall e.\, \texttt{interpret}\,(e + e) = \texttt{interpret}\, e$:

```
Test.make ~name:"test wrong"
  arb_tree
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

QCheck will refute this, but sometimes with needlessly huge, machine-generated counterexamples:

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

Test test wrong failed (0 shrink steps):

((-185826938992254398 5+-3907874316500437020)*(((-132748882265
8 340827+2616050978084075131)+12502789617362261 03)+((40060164
581 41132404+(188950103394749571 8+-9854124037720387 96))*31287
30191141124004)))
```

Shrinkers and iterators in QCheck

- A shrinker attempts to cut a counterexample down to something more comprehensible for humans
- In QCheck shrinkers are implemented as iterators: `'a Shrink.t = 'a -> 'a QCheck.Iter.t`
- Given a counterexample, QCheck will repeatedly invoke the iterator to find a simpler value, possibly still representing a counterexample
- Internally iterators are observed with `Iter.find`:

```
# let i = Iter.of_list [0;1;2;3;4;5];;
val i : int QCheck.Iter.t = <fun>
# Iter.find (fun i -> true) i;;
- : int option = Some 0
# Iter.find (fun i -> i>=3) i;;
- : int option = Some 3
# Iter.find (fun i -> i>=10) i;;
- : int option = None
```

Builtin QCheck shrinkers

`set_shrink` s g constructs a shrinking generator out of a shrinker s and a generator g.

Again `make` accepts an optional `~shrink:`s argument.

We typically build new shrinkers as a combination of iterators and by composing builtin shrinkers.

QCheck comes with a number of builtin shrinkers:

- `Shrink.nil` performs no shrinking
- `Shrink.int` for reducing integers
- `Shrink.string` for reducing strings
- `Shrink.list` for reducing lists
- `Shrink.pair` for reducing pairs
- ...

Builtin QCheck iterators

QCheck also comes with a number of builtin iterators:

- `Iter.empty` is an empty iterator
- `Iter.return` v is a one-element iterator
- `Iter.of_list` $[v_1; \ldots; v_n]$ creates an iterator out of a list of values
- `Iter.pair` i_1 i_2 creates a pair iterator out of two iterators
- `Iter.map` f g transforms an `'a` iterator g into a `'b` iterator using `f : 'a -> 'b`
- `Iter.append` i_1 i_2 sequentializes two iterators (this is also available under the infix name `<+>`)
- ...

A shrinking example (1/2)

We can hand-write a shrinker for our tree type:

```
let rec tshrink e = match e with
  | Lit i ->
  | Plus (ae0, ae1)  ->


  | Times (ae0, ae1) ->
```

A shrinking example (1/2)

We can hand-write a shrinker for our tree type:

```
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i' -> Lit i') (Shrink.int i)
  | Plus (ae0, ae1)  ->



  | Times (ae0, ae1) ->
```

For leaves, we shrink the integer and re-wrap the result

A shrinking example (1/2)

We can hand-write a shrinker for our tree type:

```
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i' -> Lit i') (Shrink.int i)
  | Plus (ae0, ae1)  ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Plus (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Plus (ae0,ae1')) (tshrink ae1)))
  | Times (ae0, ae1) ->
    Iter.append (Iter.of_list [ae0; ae1])
     (Iter.append
       (Iter.map (fun ae0' -> Times (ae0',ae1)) (tshrink ae0))
       (Iter.map (fun ae1' -> Times (ae0,ae1')) (tshrink ae1)))
```

For leaves, we shrink the integer and re-wrap the result

The inductive cases compose three iterators sequentially

A shrinking example (1/2)

We can hand-write a shrinker for our tree type:

```
let (<+>) = Iter.(<+>)    (* makes the infix operator visible *)
let rec tshrink e = match e with
  | Lit i -> Iter.map (fun i' -> Lit i') (Shrink.int i)
  | Plus (ae0, ae1)  ->
    (Iter.of_list [ae0; ae1])
    <+> (Iter.map (fun ae0' -> Plus (ae0',ae1)) (tshrink ae0))
    <+> (Iter.map (fun ae1' -> Plus (ae0,ae1')) (tshrink ae1))
  | Times (ae0, ae1) ->
    (Iter.of_list [ae0; ae1])
    <+> (Iter.map (fun ae0' -> Times (ae0',ae1)) (tshrink ae0))
    <+> (Iter.map (fun ae1' -> Times (ae0,ae1')) (tshrink ae1))
```

It actually reads a bit better using the infix-name `<+>`:

- either shrink to a sub-tree or
- recursively shrink a sub-tree and splice the resulting tree back in

A shrinking example (2/2)

We can now use our shrinker to cut down a counterexample:

```
Test.make ~name:"test wrong"
  (set_shrink tshrink arb_tree)
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

A shrinking example (2/2)

We can now use our shrinker to cut down a counterexample:

```
Test.make ~name:"test␣wrong"
  (set_shrink tshrink arb_tree)
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

with a nice effect:

```
generated error fail pass / total     time test name
[✗]     1     0    1     0 /   100     0.0s test wrong

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

Test test wrong failed (67 shrink steps):

1
```

A shrinking example (2/2)

We can now use our shrinker to cut down a counterexample:

```
Test.make ~name:"test␣wrong"
  (set_shrink tshrink arb_tree)
  (fun e -> interpret (Plus(e,e)) = interpret e)
```

with a nice effect:

```
generated error fail pass / total     time test name
[✗]     1     0    1     0 /   100     0.0s test wrong

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

Test test wrong failed (67 shrink steps):

1
```

Tip: Put the most aggressive shrinking strategy first — because iterators try things in order

Summary

We've taken a brief look at OCaml's module system

We've also seen

- how shrinkers can make a huge difference
- how QCheck comes with a number of builtin shrinkers
- how to write shrinkers
  - by expressing iterators
  - by composing iterators
  - by utilizing builtin shrinkers