02913 Advanced Analysis Techniques

QuickCheck, Day 4

Jan Midtgaard

DTU Compute

DTU

Yesterday's exercises

DTU

Lessons learned

To summarize some of our hard-learned lessons:

- 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.
- 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?

DTU

Outline

OCaml's module system

Quickchecking stateful code

Algebraic specification

Model-based specification

DTU

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,

DTU

QuickCheck, recap

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

- properties
- generators (type-directed)
- classification
- shrinking
- ...

and worked with these concepts in the `QCheck` framework

DTU

OCaml's module system

DTU

OCaml's module system (1/2)

OCaml code is typically split into modules
A module can contain both types and bindings

We can address a binding `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

DTU

OCaml's module system (1/2)

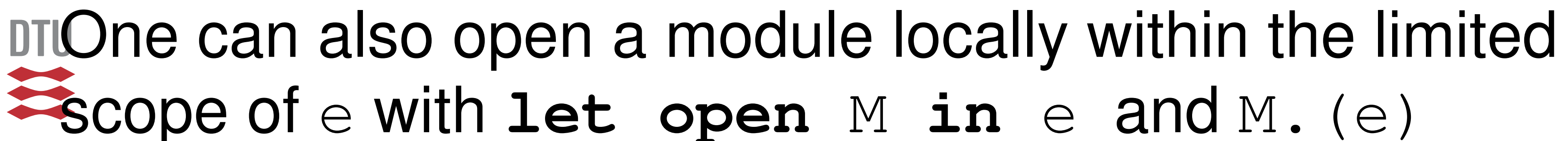
OCaml code is typically split into modules
A module can contain both types and bindings

We can address a binding `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

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

For example, we **`open`** `QCheck` at the top of our example to make the `QCheck` bindings visible



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

OCaml's module system (2/2)

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 (2/2)

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)
```

DTU

OCaml's module system (2/2)

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)
```

DTU

OCaml modules and separate compilation

We can separate the implementation and the interface of a module into two separate files `x.ml` and `x.mli`.

This is equivalent to

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

DTU

OCaml modules and separate compilation

We can separate the implementation and the interface of a module into two separate files `x.ml` and `x.mli`.

This is equivalent to

```
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`.

DTU

OCaml modules and separate compilation

We can separate the implementation and the interface of a module into two separate files `x.ml` and `x.mli`.

This is equivalent to

```
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`.

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`

DTU

Quickchecking stateful code

From functional to stateful code

For a start we've tested primarily functional code, i.e., pure code without side-effects

This setup makes life easier: functions are typically small and their functionality is determined by their parameters

But code can also be stateful: its functionality is a mixture of parameters and some internal state

QuickCheck can also be used to test such code

To do so, we should be able to generate (or bring the code to) arbitrary states and formulate properties about the state (which is today's topic)

DTU

Example: a queue

Suppose we have an imperative queue with the following interface:

```
module MyQueue :
 sig
   type 'a t                             (* the type of queues *)
   val empty : unit -> 'a t              (* queue creation *)
   val pop   : 'a t -> unit              (* may throw exception *)
   val top   : 'a t -> 'a option         (* peek at the front *)
   val push  : 'a -> 'a t -> unit        (* add element *)
 end
```

This can be easily implemented, e.g.,

- from scratch or
- as an interface to the builtin `Queue` module

DTU

Using our imperative queue

For example, we can interact with our queue at the toplevel as follows:

```
# let q = empty ();;
val q : '_a MyQueue.t = <abstr>
# push 1 q;;
- : unit = ()
# push 2 q;;
- : unit = ()
# push 3 q;;
- : unit = ()
# top q;;
- : int option = Some 1
# pop q;;
- : unit = ()
# top q;;
- : int option = Some 2
```

DTU

Algebraic specification

Our queue, algebraically

We would expect a number of algebraic equivalences:

```
let q = empty () in
 let x = top q
```
=
```
let q = empty () in
 let x = None
```

```
let q = empty () in
 let () = push m q in
  let x = top q
```
=
```
let q = empty () in
 let () = push m q in
  let x = Some m
```

```
let () = push m q in
 let () = push n q in
  let x = top q
```
=
```
let () = push m q in
 let x = top q in
  let () = push n q
```

```
let q = empty () in
 let () = push m q in
  let () = pop q
```
=
```
let q = empty ()
```

```
let () = push m q in
 let () = push n q in
  let () = pop q
```
=
```
let () = push m q in
 let () = pop q in
  let () = push n q
```

Testing imperative instructions

How can we test these properties of the queue?

Since it is an abstract data type (ADT), we can only observe it (and change it) through the provided interface

But how do we QuickCheck the interface
(the “getters” and “setters”)?

Formal semantics provides operational equivalence: two instruction sequences are indistinguishable no matter what context they are put in

So we just write a generator of all possible program contexts?

DTU

. . . almost!

Since a queue is an ADT, we can only observe it (and change it) through the official hooks

So it should be sufficient to write

- a generator of ADT contexts and
- the property of “equality of ADT observations”

We’ll generate well-formed queue instruction sequences

- that begin with `empty`,
- contains `push`, `pop`, and `top` instructions,
- any prefix contains at least as many `push` as `pop` instructions

DTU

Symbolic instructions

We can implement a symbolic type of actions:

```
type action =
    | Push of int          (* ~ let () = push n q   *)
    | Pop                  (* ~ let () = pop q      *)
    | Top                  (* ~ let x = top q       *)
    | Let of int option    (* ~ let x = None/Some n *)
```

and write an interpreter that map them to their meaning:

```
(*  interp : MyQueue.t -> action list -> int option list *)
let rec interp q acs = match acs with
  | []             -> []
  | (Push n)::acs -> begin MyQueue.push n q; interp q acs end
  | Pop::acs       -> begin MyQueue.pop q; interp q acs end
  | Top::acs       -> let e = MyQueue.top q in
                      e::(interp q acs)
  | (Let x)::acs   -> x::(interp q acs)
```

namely, a “list of observations”

From symbolic instructions to strings

QCheck (again) needs `to_string` coercion to print counterexamples:

```
(*  to_string : action -> string  *)
let to_string a = match a with
  | Push n -> "(Push␣" ^ (string_of_int n) ^ ")"
  | Pop    -> "Pop"
  | Top    -> "Top"
  | Let x  -> (match x with
               | None -> "(Let␣None)"
               | Some i -> "(Let␣(Some␣" ^ (string_of_int i)
```

Based on `to_string` we can write a version for lists of actions:

```
(*  actions_to_string : action list -> string *)
let actions_to_string = Print.list to_string
```

Number of queue elements

To only generate well-formed sequences, we need to keep track of the number of elements in a queue.

For this purpose, the following function is handy:

```
(*  delta : action list -> int  *)
let rec delta acs = match acs with
  | [] -> 0
  | (Push _)::acs' -> (delta acs') + 1
  | Pop::acs'      -> (delta acs') - 1
  | Top::acs'      -> (delta acs')
  | (Let _)::acs'  -> (delta acs')
```

It computes the element count change of a sequence of instructions

DTU

Generator, take 1

Here's a first attempt at a pure generator:

```
(*  actions : int -> action list Gen.t  *)
let rec actions num =
  Gen.oneof
    ([ Gen.return [];
       Gen.map2 (fun i acs -> (Push i)::acs)
                                  Gen.int (actions (num+1));
       Gen.map (fun acs -> Top::acs) (actions num) ]
     @ if num = 0
       then []
       else [ Gen.map (fun acs -> Pop::acs)
                                     (actions (num-1)) ])
```

It tracks the element count using a parameter `num`
and chooses between 3 or 4 different generators

DTU

Generator, take 1 – diverges!

Here's a first attempt at a pure generator:

```
(*  actions : int -> action list Gen.t  *)
let rec actions num =
  Gen.oneof
    ([ Gen.return [];
       Gen.map2 (fun i acs -> (Push i)::acs)
                                          Gen.int (actions (num+1));
       Gen.map (fun acs -> Top::acs) (actions num) ]
     @ if num = 0
       then []
       else [ Gen.map (fun acs -> Pop::acs)
                                              (actions (num-1)) ])
```

It tracks the element count using a parameter `num`

and chooses between 3 or 4 different generators

DTU

Generator, take 2 – guarded by parameter

Utilize `'a Gen.t = Random.State.t -> 'a` :

```
(*  actions : int -> action list Gen.t  *)
let rec actions num rs =
  Gen.oneof
    ([ Gen.return [];
       Gen.map2 (fun i acs -> (Push i)::acs)
                                  Gen.int (actions (num+1));
       Gen.map (fun acs -> Top::acs) (actions num) ]
     @ if num = 0
       then []
       else [ Gen.map (fun acs -> Pop::acs)
                                     (actions (num-1)) ]) rs
```

Now the recursive calls are delayed to generation time

– yet we can still use module `Gen`'s combinators

DTU

Generator, take 2 – guarded by parameter

Utilize `'a Gen.t = Random.State.t -> 'a` :

```
(*  actions : int -> action list Gen.t  *)
let rec actions num rs =
  Gen.oneof
    ([ Gen.return [];
       Gen.map2 (fun i acs -> (Push i)::acs)
                                  Gen.int (actions (num+1));
       Gen.map (fun acs -> Top::acs) (actions num) ]
     @ if num = 0
       then []
       else [ Gen.map (fun acs -> Pop::acs)
                                     (actions (num-1)) ]) rs
```

Now the recursive calls are delayed to generation time

– yet we can still use module `Gen`'s combinators

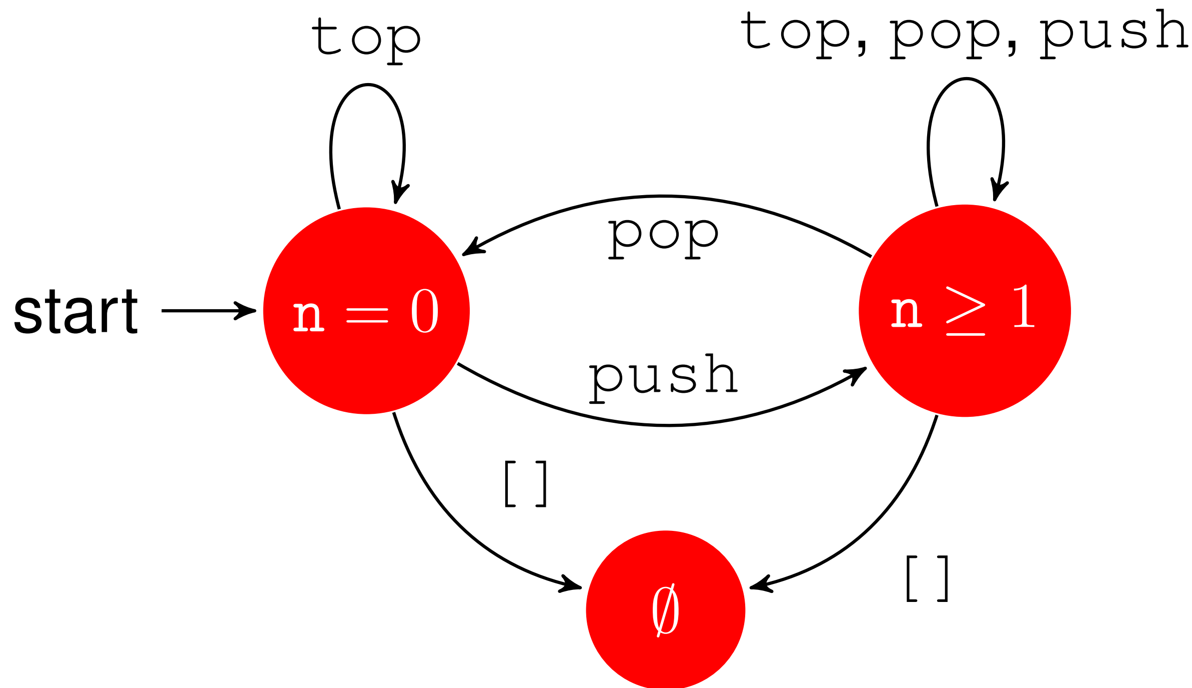
We can now write a full generator, still parameterized:

```
let arb_acts n = make ~print:actions_to_string (actions n)
```

Generator as a state machine

We can think of our generator as a state machine

(the state is the element count)



The commercial QuickCheck (for Erlang) comes with a dedicated domain specific language (or library) for writing such "state machine models"

Equivalences beginning in an empty queue

. . . we can now write tests with this function:

```
let eqempty cc' =
  let c,_ = cc' 0 in  (* hack to get c out for 'delta' call *)
  Test.make
    (pair (arb_acts (delta c)) int)
    (fun (suffix,m) ->
      let c,c' = cc' m in
      let observe is =
        let q = MyQueue.empty () in
        interp q (is @ suffix) in
      observe c = observe c')
```

which tests that two instruction sequences are observably the same with an arbitrary suffix appended.

We can now write an example test as:

```
eqempty (fun m -> [Top],[Let None])
eqempty (fun m -> [Push m; Top],[Push m; Let (Some m)])
eqempty (fun m -> [Push m; Pop],[])
```

DTU

Equivalences in any queue (1/2)

For the equivalences that hold in any queue we need generators that depend on generators

(e.g., suffix depends on prefix)

For this situation a `bind` operation is handy:

```
(>>=) : 'a Gen.t -> ('a -> 'b Gen.t) -> 'b Gen.t
```

It is written as an inline operator `>>=`

For example, we can write a generator of integer pairs, where the first component is less than the second:

```
let my_pair_gen =
  let open Gen in
  small_int >>= (fun i -> pair (int_range 0 i) (return i))
```

(`bind` also occurs in Haskell, Erlang,. . . libraries)

Equivalences in any queue (2/2)

For the equivalences that hold for any queue we can write the tests based on >>=:

```
let qprint = let open Print in
             quad actions_to_string actions_to_string int int
let eq cc' =
  let c,_ = cc' 0 0 in  (* again: get c out for delta call *)
  Test.make
    (make ~print:qprint
       (let open Gen in
        actions 0 >>= (fun pref ->
          actions (delta (pref@c)) >>= (fun suff ->
            quad (return pref) (return suff) int int))))
    (fun (pref,suff,m,n) ->
       let c,c' = cc' m n in
       let observe is =
         let q = MyQueue.empty () in
         interp q (pref @ is @ suff) in
       observe c = observe c')
```

which tests/observes under arbitrary prefix and suffix.

Equivalences in any queue (2/2)

With `eq` we can now write the last two tests as

```
eq (fun m n -> [Push m; Push n; Top],[Push m; Top; Push n;])
eq (fun m n -> [Push m; Push n; Pop],[Push m; Pop; Push n;])
```

Collectively the end result is not too far notationally from what we were after:

```
QCheck_runner.run_tests ~verbose:true [
  eqempty (fun m -> [Top],[Let None]);
  eqempty (fun m -> [Push m; Top],[Push m; Let (Some m)]);
  eqempty (fun m -> [Push m; Pop],[])
  eq (fun m n -> [Push m; Push n; Top],[Push m; Top; Push n;]);
  eq (fun m n -> [Push m; Push n; Pop],[Push m; Pop; Push n;]);
 ]
```

(it is straightforward to add an optional title argument to `eq` and `eqempty`)

Catching errors

If we introduce an error in the algebraic specification, e.g.

```
eq (fun m n -> [Push m; Push n; Pop],[Push m; Top; Push n;]);
```

QCheck will catch it:

```
law pushpop: 1 relevant cases (1 total)
  test ‘pushpop‘
  failed on ≥ 1 cases:
  ([], [Top;(Push 2018360302775497374);Pop],
                 -1435802048949759685, -1742561145930592941)
```

The counterexamples indicate that a dedicated shrinker could be useful

DTU

Model-based specification

DTU

From algebraic to model-based specifications

We just saw how to QuickCheck an imperative implementation against an algebraic specification

Next, we will study how to QuickCheck the same implementation against a model-based specification

From algebraic to model-based specifications

We just saw how to QuickCheck an imperative implementation against an algebraic specification

Next, we will study how to QuickCheck the same implementation against a model-based specification

What is a model (an abstract view) of a queue then?

From algebraic to model-based specifications

We just saw how to QuickCheck an imperative implementation against an algebraic specification

Next, we will study how to QuickCheck the same implementation against a model-based specification

What is a model (an abstract view) of a queue then?

We can easily model a queue using a list:

- the top of the queue is the list's left end
- the empty list represents the empty queue
- we push an element by appending it

DTU

Modeling a queue

This leads to the following specification (_m is for model):

```
let empty_m = []
let push_m n q = ((), q @ [n])
let pop_m q = ((), List.tl q)
let top_m q = match q with
  | []    -> (None, [])
  | n::_ -> (Some n, q)
```

All operations except `empty_m` return a pair of

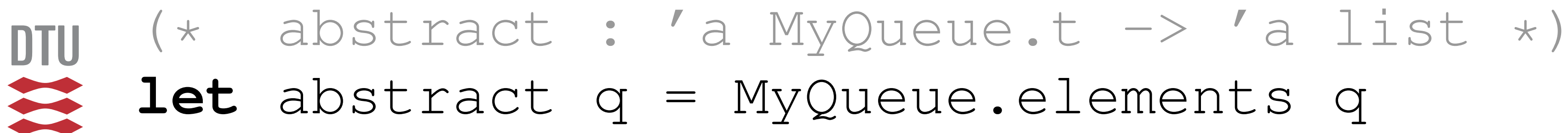
- the answer and
- the resulting queue

Relating implementation and model

We need a way to relate the implementation and the model. One way to do so is to extend the required interface slightly:

```
module MyQueue :
sig
  type 'a t
  val empty : unit -> 'a t
  val pop   : 'a t -> unit
  val top   : 'a t -> 'a option
  val push  : 'a -> 'a t -> unit
  val elements :  'a t -> 'a list
end
```

...and use the extra operation for abstracting a queue implementation to a model:



```
(*  abstract : 'a MyQueue.t -> 'a list *)
let abstract q = MyQueue.elements q
```

Warm-up: Testing `empty`

With `abstract` in hand it is straightforward to test the `empty` implementation against the specification:

```
let test_empty =
  Test.make ~name:"empty␣model" ~count:1
    unit
    (fun () ->
       let q = MyQueue.empty () in
       abstract q = empty_m)
```

In this particular case we don't need any random input!

DTU

Commuting operation and specification

Given a queue `q` along with

- an operation `op` and
- a corresponding operation over the model `model_op`

the following routine compares one against the other

```
let commutes q op model_op =
  let oldq = abstract q in
  let x = op q in
  let newq = abstract q in
  (x,newq) = model_op oldq
```

by abstracting the queue implementation before and after an operation

Writing commuting tests

We can now test commutation of implementation and model after some arbitrary operations:

```
let implements op_spec =
  Test.make
    (pair (arb_acts 0) int)
    (fun (acs,n) ->
       let op,op_m = op_spec n in
       let q = MyQueue.empty () in
       let _obs = interp q acs in
       commutes q op op_m)
```

which lets us test the `top` and `push` operations:

```
implements (fun _ -> (MyQueue.top, top_m))
implements (fun n -> (MyQueue.push n, push_m n))
```

DTU

Testing `pop` requires a precondition

We can, e.g., pass it as a parameter:

```
let implementsIf precond op_spec =
  Test.make
    (make (actions 0))
    (fun acs ->
       let q = MyQueue.empty () in
       let _obs = interp q acs in
       (precond q (* effectful operation *)
        ==> try
              let op,op_m = op_spec () in
              commutes q op op_m
            with Queue.Empty -> false))
```

and now test `pop`:

```
implementsIf (fun q -> MyQueue.top q <> None)
             (fun _ -> (MyQueue.pop, pop_m))
```

DTU

Summary

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

We've seen two approaches to test stateful code:

- One based on algebraic specifications:
  - which are tested for operational equivalence
  - requires a generator of instruction contexts
- Another based on model-based specifications:
  - which we can test the implementation against
  - also requires a generator of instruction contexts

It is helpful to view these generators as state machines

Both build on symbolic representations - and thereby permit, e.g., shrinking