

# SM2-TES: Functional Programming and Property-Based Testing, Day 6

Jan Midtgaard

MMMI, SDU

# Last week

---

- Guest lecture from MIR
- An imperative data structure in OCaml
- No time for the stateful testing part

---

# Last lecture's exercises

**Quickchecking stateful code**

**Algebraic specification**

**Model-based specification**

# 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)

# 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 (read: you did so as an exercise) or
- as an interface to the builtin `Queue` module

# Using our imperative queue

---

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

```
# let q = empty ();;
val q : 'a 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
```



# 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?

# ... 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

# Symbolic instructions

---

We can implement a **symbolic type of commands**:

```
type command =  
  | 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 : int MyQueue.t -> command list -> int option list *)  
let rec interp q cmds = match cmds with  
  | []                -> []  
  | (Push n)::cmds'  -> let () = MyQueue.push n q in  
                        interp q cmds'  
  | Pop::cmds'       -> let () = MyQueue.pop q in  
                        interp q cmds'  
  | Top::cmds'       -> let e = MyQueue.top q in  
                        e::(interp q cmds')  
  | (Let x)::cmds'   -> x::(interp q cmds')
```

namely, a **“list of observations”**

# From symbolic instructions to strings

---

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

```
(* to_string : command -> 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 commands**:

```
(* commands_to_string : command list -> string *)
let commands_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 : command list -> int *)
let rec delta cmds = match cmds with
  | [] -> 0
  | (Push _) :: cmds' -> (delta cmds') + 1
  | Pop :: cmds' -> (delta cmds') - 1
  | Top :: cmds' -> (delta cmds')
  | (Let _) :: cmds' -> (delta cmds')
```

It computes the **element count change** of a sequence of instructions

# Generator, take 1

---

Here's a first attempt at a pure generator:

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

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



# Generator, take 1 – diverges!

---

Here's a first attempt at a pure generator:

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

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

# Generator, take 2 – guarded by parameter

---

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

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

Now the **recursive calls are delayed to generation time**

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

# Generator, take 2 – guarded by parameter

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

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

Now the **recursive calls are delayed to generation time**

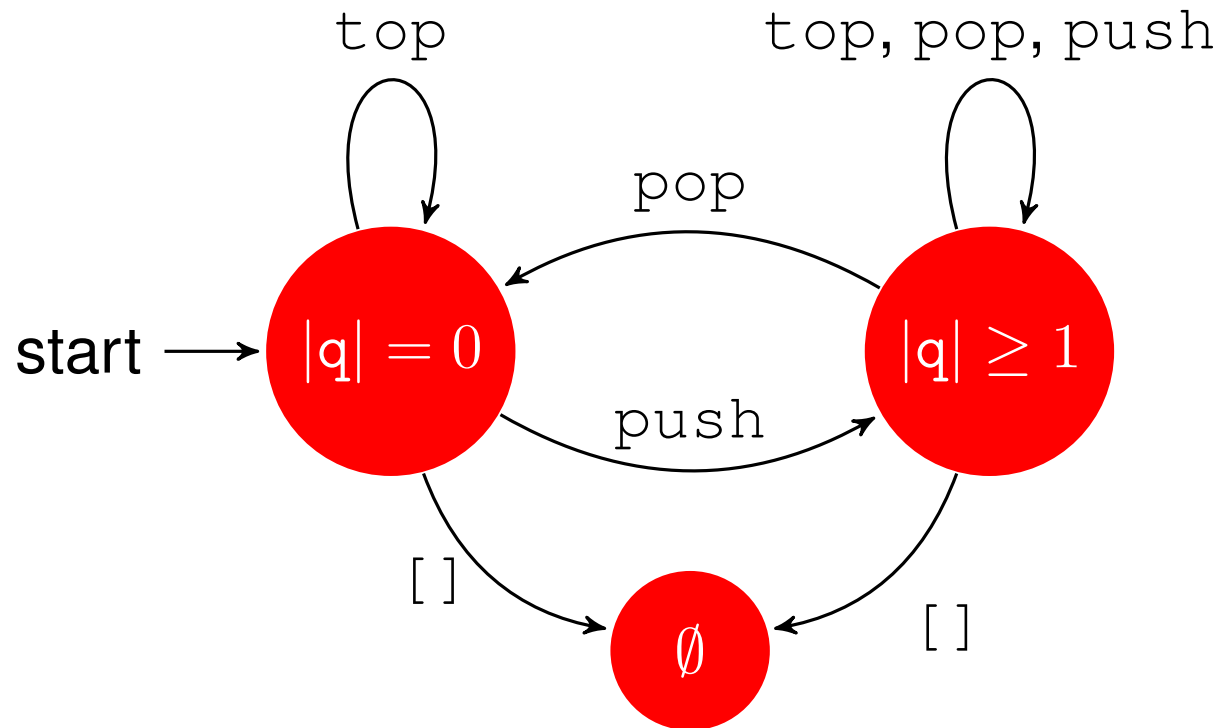
– yet we can **still use module Gen's combinators**

We can now write **a full generator**, still parameterized:

```
let arb_cmds n =
  make ~print:commands_to_string (commands n)
```

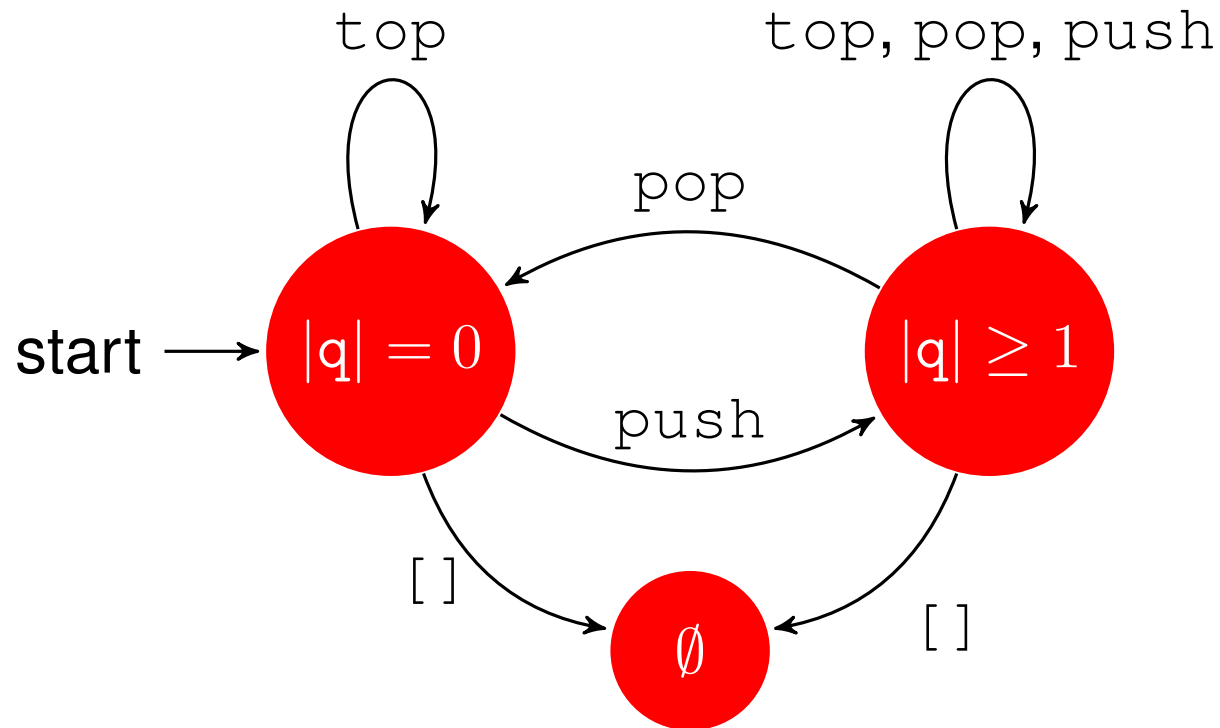
# Generator as a state machine

We can think of our generator as a state machine  
(the state is the element count)



# Generator as a state machine

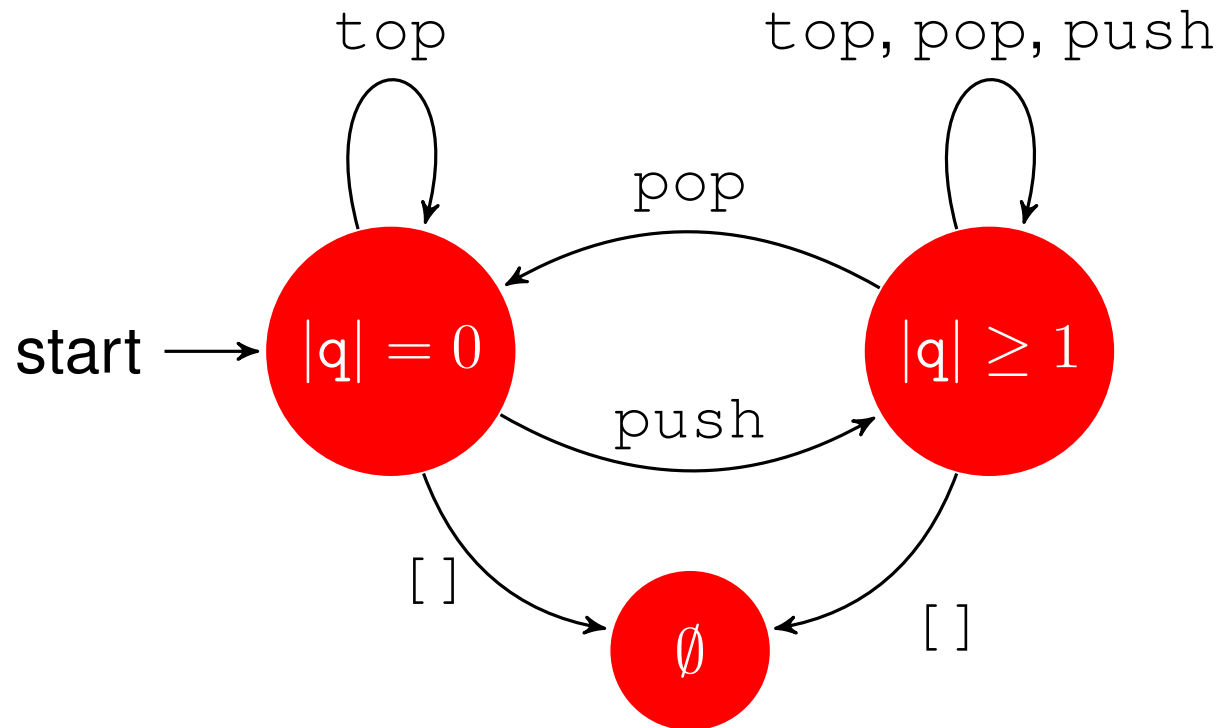
We can think of our generator as a state machine  
(the state is the element count)



With this kind of specification, `pop`'s precondition "queue is non-empty" can serve two purposes: (1) **express an implication property** and (2) **drive the generator**

# 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 cmds_pair =
  let cmds,_ = cmds_pair 0 in (*hack to get cmds for 'delta'*)
  Test.make
    (pair (arb_cmds (delta cmds)) int)
    (fun (suffix,m) ->
      let cmds,cmds' = cmds_pair m in
      let observe cmds =
        let q = MyQueue.empty () in
        interp q (cmds @ suffix) in
      observe cmds = observe cmds')
```

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

We can now write example tests as:

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

# Equivalences in any queue (1/3)

---

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 the Haskell, Erlang, . . . libraries)



# Equivalences in any queue (2/3)

---

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

```
let eq cmds_pair =
  let cmds, _ = cmds_pair 0 0 in (*again: get cmds for delta*)
  let gen_pair =
    make
      ~print:(Print.pair commands_to_string commands_to_string)
      (let open Gen in
        commands 0 >>= fun pref ->
          commands (delta (pref@cmds)) >>= fun suff ->
            return (pref,suff)) in
    Test.make (triple gen_pair int int)
    (fun ((pref,suff),m,n) ->
      let cmds,cmds' = cmds_pair m n in
      let observe cmds =
        let q = MyQueue.empty () in
        interp q (pref @ cmds @ suff) in
      observe cmds = observe cmds' )
```

SDU! which tests/observes under arbitrary prefix and suffix. 21 / 35

# Equivalences in any queue (3/3)

---

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:

```
generated error fail pass / total      time test name
```

```
...
```

```
[X]      1      0      1      0 / 1000      0.0s pushpop
```

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

```
Test pushpop failed (123 shrink steps):
```

```
(([], [Pop; Top; (Push -2942454129666812723)]), 0, 0)
```

While the last two numbers have been shrunk,

a dedicated command list shrinker could be useful

# 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**  
– akin to how we tested a functional Patricia tree implementation against a model

# 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** – akin to how we tested a functional Patricia tree implementation against a model

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** – akin to how we tested a functional Patricia tree implementation against a model

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

# 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 *)
```

# 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 *)
```

# 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!**

# 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  $res$  =  $op$   $q$  in  
  let  $newq$  = abstract  $q$  in  
  ( $res, newq$ ) =  $model\_op$   $oldq$ 
```

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 test_top =  
  Test.make ~name:"top_model"  
    (arb_cmds 0)  
    (fun cmds ->  
      let q = MyQueue.empty () in  
      let _obs = interp q cmds in  
      commutes q MyQueue.top top_m)
```

```
let test_push =  
  Test.make ~name:"push_model"  
    (pair (arb_cmds 0) int)  
    (fun (cmds,n) ->  
      let q = MyQueue.empty () in  
      let _obs = interp q cmds in  
      commutes q (MyQueue.push n) (push_m n))
```

# Testing `pop` requires a precondition

---

To test `pop` we ensure the precondition with an implication `==>` property:

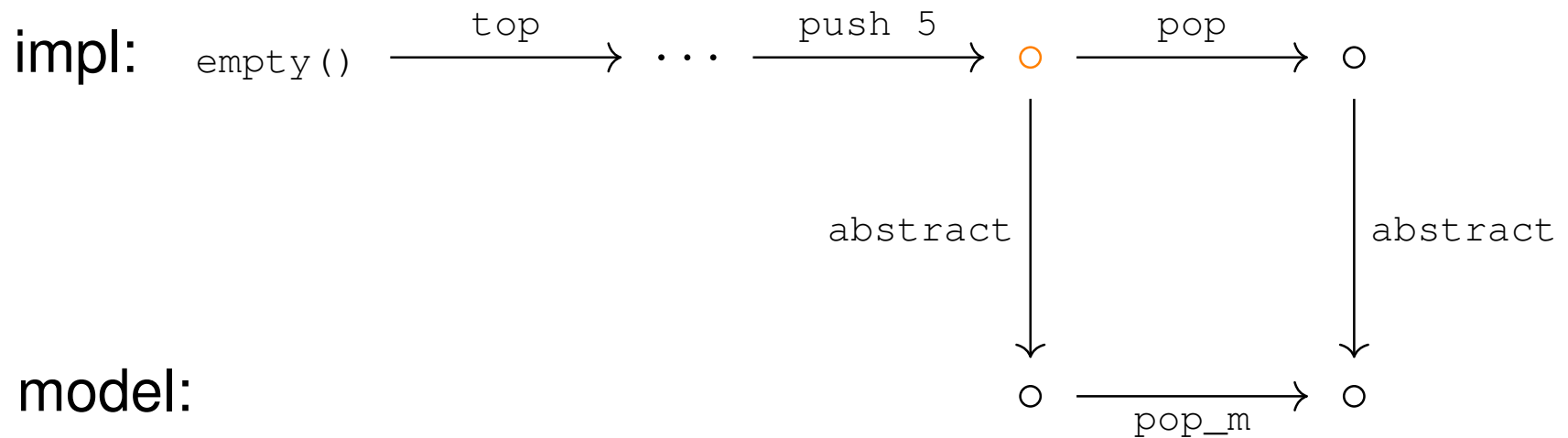
```
let test_pop =  
  Test.make ~name:"pop_model"  
    (arb_cmds 0)  
    (fun cmds ->  
      let q = MyQueue.empty () in  
      let _obs = interp q cmds in  
      (MyQueue.top q <> None  
       ==> try commutes q MyQueue.pop pop_m  
          with Queue.Empty -> false))
```

Note how the right-hand side is **protected by an exception handler**

# The big picture

---

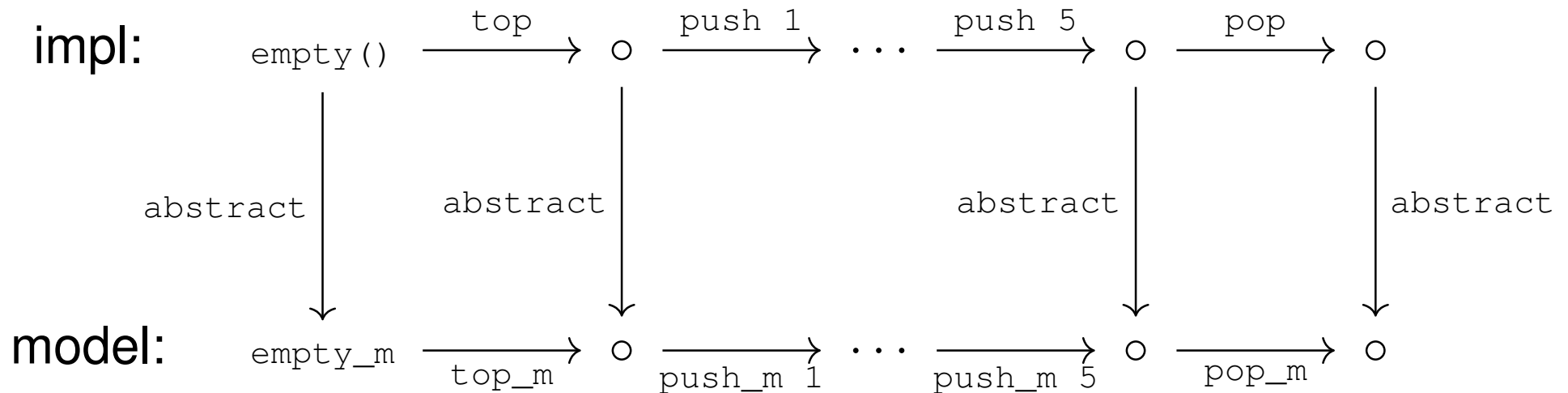
We've used an approach where generating and interpreting an arbitrary sequence of `commands` takes our model to **some state**:



At this point we then check agreement with `commutes`

# An alternative model-based approach

Alternatively, we could have checked agreement after each command in such an arbitrary sequence:



Implementation-wise this approach could be expressed in different ways.



# An implementation of the alternative approach

## Interpret commands simultaneously over impl + model:

```
(* interps_agree : command list -> int MyQueue.t -> int list -> bool *)
let rec interps_agree cmds q m = match cmds with
  | [] -> true
  | (Push n)::cmds' ->
    let () = MyQueue.push n q in
    let (),m' = push_m n m in
    abstract q = m' && interps_agree cmds' q m'
  | Pop::cmds' ->
    let () = MyQueue.pop q in
    let (),m' = pop_m m in
    abstract q = m' && interps_agree cmds' q m'
  | Top::cmds' ->
    let opt = MyQueue.top q in
    let opt',m' = top_m m in
    opt = opt' && abstract q = m && interps_agree cmds' q m'
  | (Let x)::cmds' ->
    interps_agree cmds' q m
```

# An implementation of the alternative approach

## Interpret commands simultaneously over impl + model:

```
(* interps_agree : command list -> int MyQueue.t -> int list -> bool *)
let rec interps_agree cmds q m = match cmds with
| [] -> true
| (Push n)::cmds' ->
  let () = MyQueue.push n q in
  let (),m' = push_m n m in
  abstract q = m' && interps_agree cmds' q m'
| Pop::cmds' ->
  let () = MyQueue.pop q in
  let (),m' = pop_m m in
  abstract q = m' && interps_agree cmds' q m'
| Top::cmds' ->
  let opt = MyQueue.top q in
  let opt',m' = top_m m in
  opt = opt' && abstract q = m && interps_agree cmds' q m'
| (Let x)::cmds' ->
  interps_agree cmds' q m
```

## Using interps\_agree we can now test this property:

```
Test.make ~name:"alternative_agreement" ~count:1000
  (arb_cmds 0)
  (fun cmds -> let q = MyQueue.empty () in
               let m = empty_m in
               abstract q = m && interps_agree cmds q m)
```

# Summary

---

We've seen two approaches to test stateful code:

- One based on **algebraic specifications**:
  - which are tested for operational equivalence
  - requires a generator of command contexts
- Another based on **model-based specifications**:
  - which we can test the implementation against
  - also requires a generator of commands
  - we can test agreement in the end or at each step

It is helpful to view these **generators as state machines**

Both build on **symbolic representations**

– and thereby permit, e.g., shrinking