SM2-TES: Functional Programming and Property-Based Testing, Day 5

Jan Midtgaard

MMMI, SDU

SDU

Last week

- A brief look at OCaml's module system
- Shrinking counterexamples
- Model-based testing (of Patricia trees)

Last lecture's exercises

QuickCheck, recap

QuickCheck-wise we've covered:

- properties
- generators (type-directed)
- statistics (observing generator distributions)
- shrinking
- model-based testing (of functional data structures)
- ...

and worked with these concepts in the `QCheck` framework

Auto-deriving Boilerplate Code

Tired of writing printers?

Let the computer derive them automatically!
This works using an annotation and a preprocessor.

First install the preprocessor:

```
opam install ppx_deriving
```

Then in utop:

```
# #require "ppx_deriving.show";;
# type color = Red | Blue | Green [@@deriving show];;
type color = Red | Blue | Green
val pp_color : Format.formatter -> color -> unit = <fun>
val show_color : color -> string = <fun>
# show_color Red;;
- : string = "Red"
```

So from a simple type annotation we get two functions for free! (You can also derive `equal`, `compare`, ...)

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 depends on both 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: hashtables

Consider the hashtables from OCaml's standard library:

```
val create : ?random:bool -> int -> ('a, 'b) Hashtbl.t
val add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit
val remove : ('a, 'b) Hashtbl.t -> 'a -> unit
val find : ('a, 'b) Hashtbl.t -> 'a -> 'b
val mem : ('a, 'b) Hashtbl.t -> 'a -> bool
```

`('a, 'b) Hashtbl.t` is a parametric type where

- `'a` ranges over the key type
- `'b` ranges over the value type

For example, `(string, int) Hashtbl.t` represents hashtables mapping `string`s to `int`s

Using the hashtables

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

```
# let h = Hashtbl.create 5;;
val h : ('_weak3, '_weak4) Hashtbl.t = <abstr>
# Hashtbl.add h "a" 4;;
- : unit = ()
# Hashtbl.add h "z" 0;;
- : unit = ()
# Hashtbl.mem h "a";;
- : bool = true
# Hashtbl.remove h "a";;
- : unit = ()
# Hashtbl.find h "a";;
Exception: Not_found.
#
```

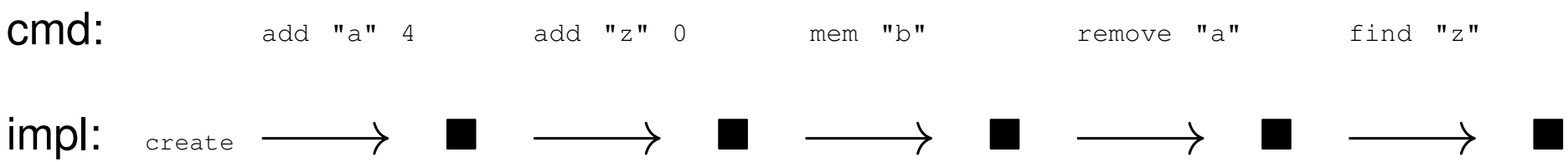
From random inputs to random commands

Rather than extract the internal state of hashtables with an `abstract` operation, we will just compare outputs with a model across arbitrary command sequences:

```
add "a" 4       add "z" 0       mem "b"         remove "a"      find "z"
```

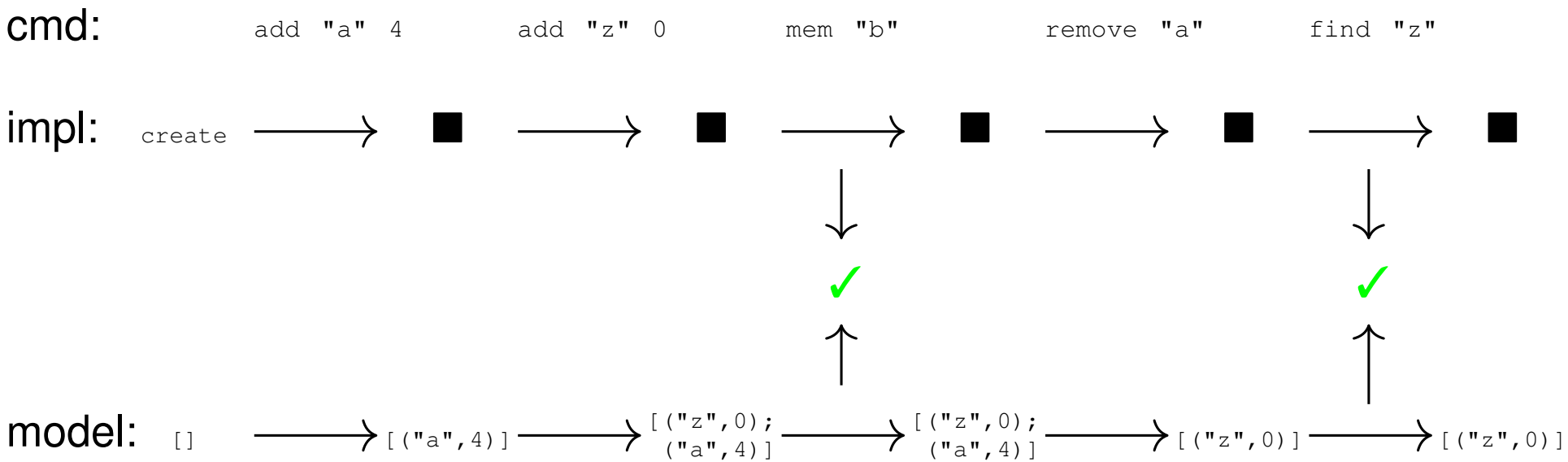
From random inputs to random commands

Rather than extract the internal state of hashtables with an `abstract` operation, we will just compare outputs with a model across arbitrary command sequences:



From random inputs to random commands

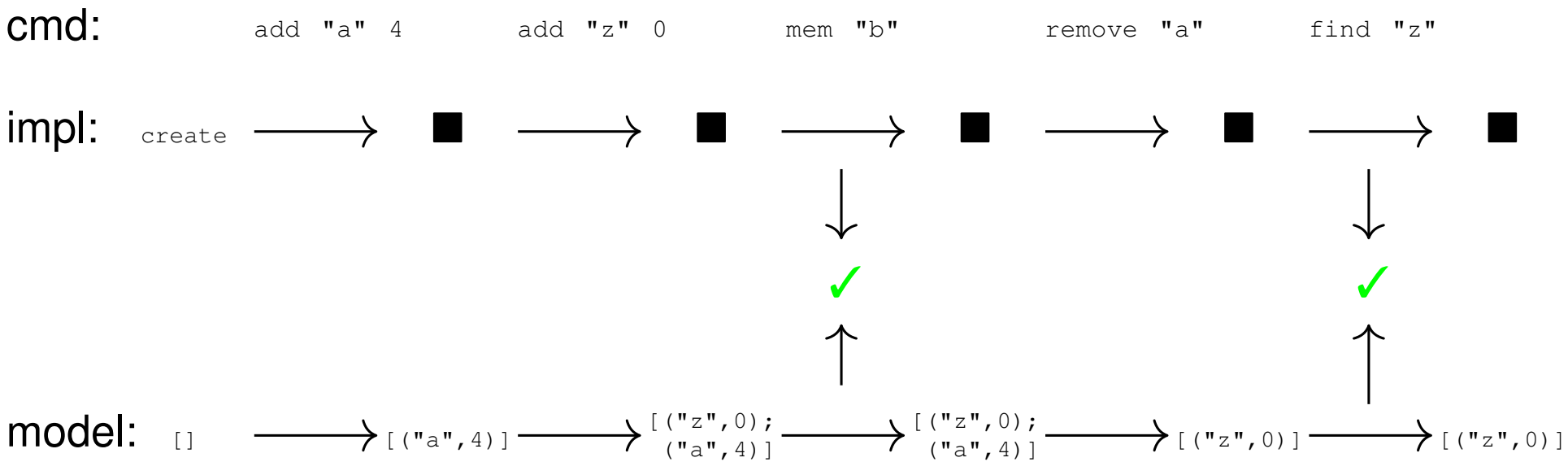
Rather than extract the internal state of hashtables with an `abstract` operation, we will just compare outputs with a model across arbitrary command sequences:



Only `mem` and `find` have (meaningful) output to compare

From random inputs to random commands

Rather than extract the internal state of hashtables with an `abstract` operation, we will just compare outputs with a model across arbitrary command sequences:



Only `mem` and `find` have (meaningful) output to compare

Bonus: we lift randomness to sequences of operations and test their interaction

Commands

To test the hashtables we need to choose concrete types for keys and values.

A(n admittedly arbitrary) type choice:

keys: `string`, values: `int`

Commands

To test the hashtables we need to choose concrete types for keys and values.

A(n admittedly arbitrary) type choice:

keys: `string`, values: `int`

Based on this choice we can represent hashtable commands symbolically:

```
type cmd =
  | Add of string * int
  | Remove of string
  | Find of string
  | Mem of string [@@deriving show { with_path = false }]
```

The `@@deriving` annotation automatically derives a function `show_cmd : cmd -> string` from the type definition.

Command generation (1/2)

We can now write a `cmd` generator:

```
(*  gen_cmd : cmd Gen.t  *)
let gen_cmd =
  let str_gen = Gen.oneof [Gen.small_string;
                           Gen.string] in
  Gen.oneof
    [ Gen.map2 (fun k v -> Add (k,v)) str_gen Gen.small_nat;
      Gen.map  (fun k   -> Remove k) str_gen;
      Gen.map  (fun k   -> Find k) str_gen;
      Gen.map  (fun k   -> Mem k) str_gen; ]
```

Here we reuse a string generator, that either

- generates a short string with `Gen.small_string`
- generates an arbitrary string with `Gen.string`

Command generation (2/2)

We can now combine the derived `show_cmd` and our pure generator into a full generator:

```
(*  arb_cmd : cmd QCheck.arbitrary *)
let arb_cmd = make ~print:show_cmd gen_cmd
```

Command generation (2/2)

We can now combine the derived `show_cmd` and our pure generator into a full generator:

```
(*  arb_cmd : cmd QCheck.arbitrary *)
let arb_cmd = make ~print:show_cmd gen_cmd
```

We subsequently lift the full generator to generate lists of `cmd`s:

```
(*  arb_cmds : cmd list QCheck.arbitrary *)
let arb_cmds = list arb_cmd
```

Command generation (2/2)

We can now combine the derived `show_cmd` and our pure generator into a full generator:

```
(*  arb_cmd : cmd QCheck.arbitrary *)
let arb_cmd = make ~print:show_cmd gen_cmd
```

We subsequently lift the full generator to generate lists of `cmd`s:

```
(*  arb_cmds : cmd list QCheck.arbitrary *)
let arb_cmds = list arb_cmd
```

It seems to work:

```
# Gen.generate1 arb_cmds.gen;;
- : cmd list =
[Remove ",C\025?.";
 Remove "\014'\026B";
 Add ("ka?KeAXPv", 6233)]
```

Modeling hashtables

Next, we need to model the imperative hashtable

– akin to how we modeled integer sets last time

Modeling hashtables

Next, we need to model the imperative hashtable

– akin to how we modeled integer sets last time

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

Modeling hashtables

Next, we need to model the imperative hashtable

– akin to how we modeled integer sets last time

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

We can easily model a hashtable using a list of pairs:

```
type state = (string * int) list
```

- each pair represents a key-value entry
- the empty list represents the empty hashtable
- we add a new entry by appending it
- if two pairs have the same key, the left shadows the right

This is also called an association list

Interpreting commands over the model

It is now straightforward to write an `cmd`-interpreter over our model's state:

```
(*  next_state : cmd -> state -> state *)
let next_state c s = match c with
  | Add (k,v) -> (k,v)::s
  | Remove k  -> List.remove_assoc k s
  | Find _
  | Mem _     -> s
```

- `Find` and `Mem` do not change the hashtable's state.
- `Add` inserts an entry on the right
- `Remove` relies on

  ```
  remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
  ```

  from the standard library.

Interpreting commands over hashtables

Now, we need to

- interpret commands over the actual hashtable
- ensure agreement with the model

Interpreting commands over hashtables

Now, we need to

- interpret commands over the actual hashtable
- ensure agreement with the model

Both tasks are handled by `run_cmd`:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> bool*)
let run_cmd c s h = match c with
  | Add (k,v) -> begin Hashtbl.add h k v; true end
  | Remove k  -> begin Hashtbl.remove h k; true end
  | Find k    ->
    List.assoc_opt k s = (try Some (Hashtbl.find h k)
                          with Not_found -> None)
  | Mem k     -> List.mem_assoc k s = Hashtbl.mem h k
```

- The `Add` and `Remove` cases have no output to check
- The `Find` and `Mem` cases check agreement with the model

Putting the pieces together (1/2)

We can now write a recursive agreement checker:

```
(*  interp_agree :
      state -> (string, int) Hashtbl.t -> cmd list -> bool *)
let rec interp_agree s h cs = match cs with
  | [] -> true
  | c::cs ->
    let b = run_cmd c s h in
    let s' = next_state c s in
    b && interp_agree s' h cs
```

- Each command is first interpreted over the hashtable and then over the model.
- In case of disagreement we stop the recursion early (thanks to short-cut `bool` evaluation of `&&`)

Putting the pieces together (2/2)

We can now phrase a QCheck test:

```
(*  agree_test : QCheck.Test.t  *)
let agree_test =
  Test.make ~name:"Hashtbl-model␣agreement" ~count:500
    arb_cmds
    (fun cs ->
       interp_agree [] (Hashtbl.create ~random:false 42) cs)
```

starting from the empty association list and an empty hashtable.

If we run it QCheck finds no disagreement:

```
random seed: 106718569
generated error fail pass / total     time test name
[✓]  500     0    0  500 /  500     1.9s Hashtbl-model agreement
============================================================
success (ran 1 tests)
```

So can we assume everything is fine?

Intermezzo: Fault Injection

Fault injection

How can we be sure that our tests work as expected?

Fault injection

How can we be sure that our tests work as expected?

One way to “test the tester” is by fault injection:

- introduce a deliberate error (in model or hashtable)
- check whether it is caught

Fault injection

How can we be sure that our tests work as expected?

One way to "test the tester" is by fault injection:

- introduce a deliberate error (in model or hashtable)
- check whether it is caught

If caught: hurrah!

(perhaps retry by injecting another error?)

If not caught: oops! Why not? Revise test suite?

Fault injection

How can we be sure that our tests work as expected?

One way to "test the tester" is by fault injection:

- introduce a deliberate error (in model or hashtable)
- check whether it is caught

If caught: hurrah!

(perhaps retry by injecting another error?)

If not caught: oops! Why not? Revise test suite?

Fault injection is useful outside property-based testing

(actually, it was invented before. . .)

It is a useful indicator of a test suite's strength

Testing our tester. . . (1/2)

Suppose we change `run_cmd` as follows:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> bool*)
let run_cmd c s h = match c with
  | Add (k,v) -> begin Hashtbl.add h k (v+1); true end
  (* remaining cases left unmodified *)
```

This error simply inserts `v+1` rather than `v` in the hashtable.

Testing our tester. . . (1/2)

Suppose we change `run_cmd` as follows:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> bool*)
let run_cmd c s h = match c with
  | Add (k,v) -> begin Hashtbl.add h k (v+1); true end
  (* remaining cases left unmodified *)
```

This error simply inserts `v+1` rather than `v` in the hashtable.

It is immediately caught by our model-based tests:

```
random seed: 489150886
generated error fail pass / total     time test name
[✗]     9     0    1     8 /  500     0.0s Hashtbl-model agreement

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

Test Hashtbl-model agreement 1 failed (8 shrink steps):

[(Add ("", 74)); (Find "")]
```

Hurrah! It found a short, understandable counterexample

Testing our tester. . . (2/2)

Suppose we instead change `run_cmd` as follows:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> bool*)
let run_cmd c s h = match c with
  | Add (k,v) ->
    if String.length k <= 2
    then begin Hashtbl.add h k v; true end
    else begin Hashtbl.add h k (v+1); end
  (* remaining cases left unmodified *)
```

When the key's length > 2 we insert `v+1` rather than `v`

Somewhat surprisingly, this error is not caught:

```
random seed: 326199985
generated error fail pass / total     time test name
[✓]  500     0    0  500 /  500     1.9s Hashtbl-model agreement
============================================================
success (ran 1 tests)
```

Testing our tester. . . (2/2)

Suppose we instead change `run_cmd` as follows:

```
(*  run_cmd : cmd -> state -> (string, int) Hashtbl.t -> bool*)
let run_cmd c s h = match c with
  | Add (k,v) ->
    if String.length k <= 2
    then begin Hashtbl.add h k v; true end
    else begin Hashtbl.add h k (v+1); end
  (* remaining cases left unmodified *)
```

When the key's length > 2 we insert `v+1` rather than `v`

Somewhat surprisingly, this error is not caught:

```
random seed: 326199985
generated error fail pass / total     time test name
[✓]   500     0    0   500 /   500     1.9s Hashtbl-model agreement
============================================================
success (ran 1 tests)
```

Why is this so? Hmmmm. . .

The problem

To catch this injected error a counterexample needs

- an insertion with a key longer than 2, e.g., `Add ("abc", 41)`
- a later lookup of the same key, e.g., `Find "abc"`

Q: Why won't `Remove "abc"` or `Mem "abc"` do instead of `Find "abc"`?

The problem

To catch this injected error a counterexample needs

- an insertion with a key longer than 2, e.g., `Add ("abc", 41)`
- a later lookup of the same key, e.g., `Find "abc"`

Q: Why won't `Remove "abc"` or `Mem "abc"` do instead of `Find "abc"`?

Our string generator has a low probability of producing identical strings longer than 2 characters.

Low chance of such output means low chance of catching the error...

Solution: state-dependent generation

Essentially the generator should mimic hand-written tests: sometimes look up an earlier added key. . .

We write such a generator by passing the model's state:

```
(*  gen_cmd : state -> cmd Gen.t  *)
let gen_cmd s =
  let str_gen =
    if s = []
    then Gen.oneof [Gen.small_string;
                    Gen.string]
    else
      let keys = List.map fst s in
      Gen.oneof [Gen.oneofl keys;
                 Gen.small_string;
                 Gen.string] in
  Gen.oneof
    [ Gen.map2 (fun k v -> Add (k,v)) str_gen Gen.small_nat;
      Gen.map  (fun k   -> Remove k) str_gen;
      Gen.map  (fun k   -> Find k) str_gen;
      Gen.map  (fun k   -> Mem k) str_gen; ]
```

This has $\frac{1}{3}$ chance of generating an existing entry (when the model is non-empty)

A ‘bind’-operation for the toolbox

To generate state-dependent `cmd list`s we need generators that depend on generators

For this situation a `bind` operation is handy:

```
Gen.(>>=) : '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_nat >>= fun i -> pair (int_bound i) (return i) in
  Gen.generate ~n:4 my_pair_gen;;
- : (int * int) list = [(0, 6); (40, 67); (4, 9); (0, 6)]
```

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

Lists of commands, dependently

With `Gen.(>>=)` we can now write a state-dependent generator of `cmd list`s:

```
(*  gen_cmds : state -> int -> cmd list QCheck.Gen.t *)
let rec gen_cmds s fuel =
  let open Gen in
  if fuel = 0
  then Gen.return []
  else
    gen_cmd s >>= fun c ->
      (gen_cmds (next_state c s) (fuel-1)) >>= fun cs ->
        return (c::cs)
```

- If we are out of fuel: we generate `[]`
- If we still have fuel:
  generate a command, name it `c`
  generate a tail (recursively), name it `cs`
  glue them together and return as generator result

Testing the state-dependent generator

We can now revise the full generator, where we start it off from the empty association list `[]`:

```
(*  arb_cmds : cmd list QCheck.arbitrary *)
let arb_cmds =
  make ~print:(Print.list show_cmd) ~shrink:Shrink.list
    (Gen.sized (gen_cmds []))
```

Testing the state-dependent generator

We can now revise the full generator, where we start it off from the empty association list `[]`:

```
(*  arb_cmds : cmd list QCheck.arbitrary *)
let arb_cmds =
  make ~print:(Print.list show_cmd) ~shrink:Shrink.list
    (Gen.sized (gen_cmds []))
```

The revised version quickly catches the injected error:

```
random seed: 136017840
generated error fail pass / total     time test name
[✗]    6    0    1    5 /  500     0.0s Hashtbl-model agreement

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

Test Hashtbl-model agreement 2 failed (10 shrink steps):

[(Add ("^\228\203P", 5)); (Find "^\228\203P")]
================================================================
```

After only 6 attempts, it found a minimal counterexample of only 2 commands.

From Hashtables to Stateful Systems in General

State machine models in general

Generalizing from the hashtable example, we need:

- a type of commands
- a system under test (hashtables)
- a model of the system's state (association lists)

State machine models in general

Generalizing from the hashtable example, we need:

- a type of commands
- a system under test (hashtables)
- a model of the system's state (association lists)

and operations for

- interpreting commands over the model
- interpreting commands over the system under test and assuring agreement
- ensuring agreement over a list of commands

State machine models in general

Generalizing from the hashtable example, we need:

- a type of commands
- a system under test (hashtables)
- a model of the system's state (association lists)

and operations for

- interpreting commands over the model
- interpreting commands over the system under test and assuring agreement
- ensuring agreement over a list of commands

Furthermore, state-dependent command generation can be useful.

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd
  type state
  type sut

  val arb_cmd : state -> cmd arbitrary

  val init_state : state
  val next_state : cmd -> state -> state

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state
  type sut

  val arb_cmd : state -> cmd arbitrary

  val init_state : state
  val next_state : cmd -> state -> state

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut

  val arb_cmd : state -> cmd arbitrary

  val init_state : state
  val next_state : cmd -> state -> state

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary

  val init_state : state
  val next_state : cmd -> state -> state

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state
  val next_state : cmd -> state -> state

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state                (* initialize model *)
  val next_state : cmd -> state -> state

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state                (* initialize model *)
  val next_state : cmd -> state -> state (* model int. *)

  val init_sut : unit -> sut
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state                (* initialize model *)
  val next_state : cmd -> state -> state (* model int. *)

  val init_sut : unit -> sut            (* initialize sut *)
  val cleanup : sut -> unit
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state                (* initialize model *)
  val next_state : cmd -> state -> state (* model int. *)

  val init_sut : unit -> sut              (* initialize sut *)
  val cleanup : sut -> unit                  (* reset sut *)
  val run_cmd : cmd -> state -> sut -> bool

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state                (* initialize model *)
  val next_state : cmd -> state -> state (* model int. *)

  val init_sut : unit -> sut              (* initialize sut *)
  val cleanup : sut -> unit                  (* reset sut *)
  val run_cmd : cmd -> state -> sut -> bool (* run sut *)

  val precond : cmd -> state -> bool
end
```

A common signature: `StmSpec`

We can capture these commonalities in a signature:

```
module type StmSpec =
sig
  type cmd     (* the type of commands *)
  type state   (* the model's state *)
  type sut     (* type of the system under test *)

  val arb_cmd : state -> cmd arbitrary (* full cmd gen *)

  val init_state : state                (* initialize model *)
  val next_state : cmd -> state -> state (* model int. *)

  val init_sut : unit -> sut              (* initialize sut *)
  val cleanup : sut -> unit                  (* reset sut *)
  val run_cmd : cmd -> state -> sut -> bool (* run sut *)

  val precond : cmd -> state -> bool
end
```

We will discuss `precond` shortly...

An example instance of `StmSpec`

```
module HConf =
struct
  type state = (string * int) list
  type sut   = (string, int) Hashtbl.t
  type cmd =
    | Add of string * int
    | Remove of string
    | Find of string
    | Mem of string [@@deriving show { with_path = false }]

  (*  gen_cmd : state -> cmd Gen.t  *)
  let gen_cmd s = (* ... as before *)
  let arb_cmd s = QCheck.make ~print:show_cmd (gen_cmd s)

  let init_state  = []
  let next_state c s = (* ... as before *)

  let init_sut () = Hashtbl.create ~random:false 42
  let cleanup _   = ()
  let run_cmd c s h =  (* ... as before *)

  let precond _ _ = true
end
```

This satisfies `StmSpec` with the parts we have seen...

QCSTM: a state-machine framework

QCSTM is a little state-machine framework I wrote.

It has a functor `QCSTM.Make`, that expects the `StmSpec` signature and outputs a module with this signature:

```
sig
  (* some entries omitted *)
  val arb_cmds : Spec.state -> Spec.cmd list arbitrary
  val interp_agree : Spec.state -> Spec.sut -> Spec.cmd list -> bool
  val agree_test : ?count:int -> name:string -> Test.t
end
```

It produces, e.g., a state-dependent `cmd list` generator and a recursive agreement checker.

QCSTM: a state-machine framework

QCSTM is a little state-machine framework I wrote.

It has a functor `QCSTM.Make`, that expects the `StmSpec` signature and outputs a module with this signature:

```
sig
  (* some entries omitted *)
  val arb_cmds : Spec.state -> Spec.cmd list arbitrary
  val interp_agree : Spec.state -> Spec.sut -> Spec.cmd list -> bool
  val agree_test : ?count:int -> name:string -> Test.t
end
```

It produces, e.g., a state-dependent `cmd list` generator and a recursive agreement checker.

With `QCSTM` we can run the hashtable state-machine model test as follows:

```
module HT = QCSTM.Make(HConf)
;;
QCheck_runner.run_tests ~verbose:true
  [HT.agree_test ~count:500 ~name:"Hashtbl-model␣agreement"]
```

Another Example: Queues

Example: queues

Consider the `Queue` module from the standard library.

It has the following signature:

```
module Queue :
 sig
   type 'a t                              (* the type of queues *)
   val create : unit -> 'a t              (* queue creation *)
   val pop    : 'a t -> 'a                (* may throw exception *)
   val top    : 'a t -> 'a                (* may throw exception *)
   val push   : 'a -> 'a t -> unit        (* add element *)
   (* other bindings omitted *)
 end
```

The API implements FIFO (first-in, first-out) queues as you would expect.

Using imperative queues

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

```
# let q = Queue.empty ();;
val q : '_weak1 Queue.t = <abstr>
# Queue.top q;;
Exception: Stdlib.Queue.Empty.
# Queue.push 1 q;;
- : unit = ()
# Queue.push 2 q;;
- : unit = ()
# Queue.push 3 q;;
- : unit = ()
# Queue.pop q;;
- : int = 1
# Queue.top q;;
- : int = 2
```

Setting up the types. . .

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

If we furthermore fix the element type to `int`,
the three types are in place:

```
type state = int list
type sut = int Queue.t
type cmd =
  | Pop
  | Top
  | Push of int [@@deriving show { with_path = false }]
```

Generating commands

Again we phrase a state-dependent generator:

```
let gen_cmd s =
  let int_gen = Gen.small_nat in
  if s = []
  then Gen.map (fun i -> Push i) int_gen
  else Gen.oneof
         [Gen.return Pop;
          Gen.return Top;
          Gen.map (fun i -> Push i) int_gen]
```

From an empty queue we can only generate `Push`es

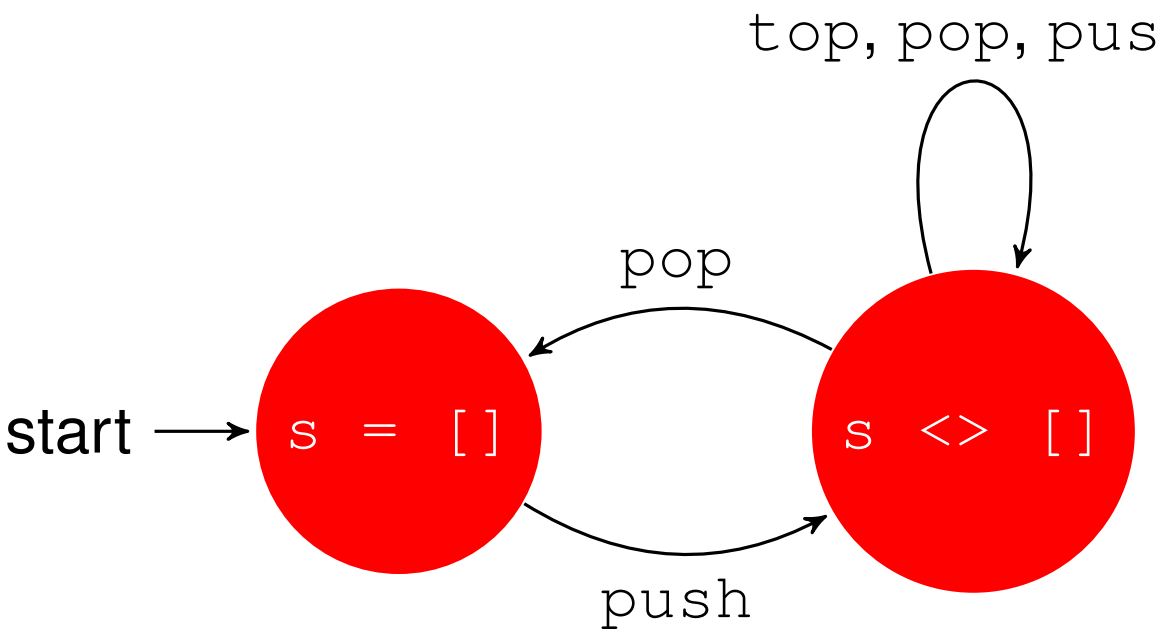
From a non-empty queue we can generate either `Pop`, `Top`, or `Push`

We can lift it to a full state-dependent generator:

```
let arb_cmd s = QCheck.make ~print:show_cmd (gen_cmd s)
```

The generator as a state machine

We can think of our generator as a state machine:



This way, we ensure that it only produces `cmd lists` that do not raise exceptions.

Once the shrinker starts removing `cmd`s this invariant may unfortunately be broken...

Modeling the queue

An empty queue is modeled by an empty list:

```
let init_state = []
```

Similarly to the hashtable example, we can interpret commands over the model's state:

```
let next_state c s = match c with
  | Pop ->
    (match s with
      | []    -> [] (* sut raises exc, state is unchanged *)
      | _::s' -> s')
  | Top -> s
  | Push i -> s@[i]
```

`Pop` removes the head of the list,
`Push` adds an entry on the right, while
`Top` does not change the state

The last part: The system under test

We initialize the system by creating a new queue:

```
let init_sut () = Queue.create ()
let cleanup _   = ()
```

Queues do not need cleaning up

(but sockets, filehandles, or databases might)

The last part: The system under test

We initialize the system by creating a new queue:

```
let init_sut () = Queue.create ()
let cleanup _   = ()
```

Queues do not need cleaning up

(but sockets, filehandles, or databases might)

Finally we can interpret commands over queues:

```
let run_cmd c s q = match c with
  | Pop -> (try Queue.pop q = List.hd s with _ -> false)
  | Top -> (try Queue.top q = List.hd s with _ -> false)
  | Push n -> begin Queue.push n q; true end
```

If either `Queue.pop`, `Queue.top`, or `List.hd` raises an exception, we consider the test failed

Testing the queues (1/2)

Let's start without any preconditions:

```
(*  precond : cmd -> state -> bool *)
let precond _ _ = true
```

Our model-based queue test seems to work:

```
random seed: 217634893
generated error  fail  pass / total     time test name
[✓] 10000     0     0 10000 / 10000     1.4s queue-model agreement
```

Testing the queues (1/2)

Let's start without any preconditions:

```
(*  precond : cmd -> state -> bool *)
let precond _ _ = true
```

Our model-based queue test seems to work:

```
random seed: 217634893
generated error  fail  pass / total     time test name
[✓] 10000     0     0 10000 / 10000     1.4s queue-model agreement
```

Again we can inject an error (ignoring inserts of 98):

```
let next_state c s = match c with
  | Pop ->
    (match s with
      | []    -> []
      | _::s' -> s')
  | Top -> s
  | Push i ->
    if i<>98 then s@[i] else s   (* an artificial fault *)
```

Testing the queues (2/2)

Running it now reveals a problem:

```
random seed: 226748961
generated error  fail  pass / total     time test name
[✗]     9     0     1     8 / 10000     0.0s queue-model agreement

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

Test queue-model agreement failed (14 shrink steps):

[Pop]
```

It took only 9 tests to find a counterexample!

Unfortunately it was then shrunk (in 14 steps) into a test case we consider invalid. . .

Since our generator can never produce sequences that pop an empty queue, neither should the shrinker.

Testing the queues (2/2)

Running it now reveals a problem:

```
random seed: 226748961
generated error  fail  pass / total     time test name
[✗]     9     0     1     8 / 10000     0.0s queue-model agreement

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

Test queue-model agreement failed (14 shrink steps):

[Pop]
```

It took only 9 tests to find a counterexample!

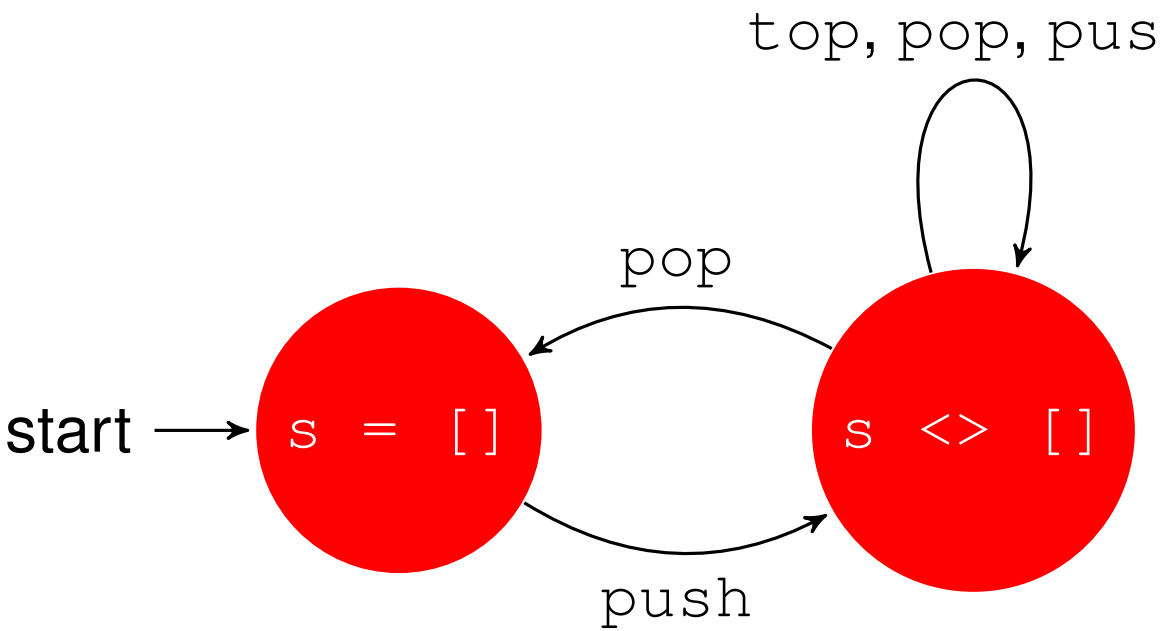
Unfortunately it was then shrunk (in 14 steps) into a test case we consider invalid. . .

Since our generator can never produce sequences that pop an empty queue, neither should the shrinker.

`precond` can guard against this

Preconditions as a state machine

We can also think of preconditions as a state machine:




```
(*  precond : cmd -> state -> bool *)
let precond c s = match c with
  | Pop    -> s <> []
  | Top    -> s <> []
  | Push _ -> true
```

Intuition: is command `c` OK in state `s`?

This way `precond` lets the shrinker distinguish between acceptable and unacceptable `cmd list`s

Testing the queues, take 2

Again we inject the error (ignoring inserts of 98):

```
let next_state c s = match c with
  | Push i ->
    if i<>98 then s@[i] else s
    (* remaining cases unchanged *)
```

Testing the queues, take 2

Again we inject the error (ignoring inserts of 98):

```
let next_state c s = match c with
  | Push i ->
    if i<>98 then s@[i] else s
    (* remaining cases unchanged *)
```

It is now caught and shrunk as expected:

```
random seed: 388881186
generated error  fail  pass / total     time test name
[✗]    42     0     1    41 / 10000     0.6s queue-model agreement

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

Test queue-model agreement failed (11 shrink steps):

[(Push 60); (Push 98); (Push 8); Pop; Top]
```

Testing the queues, take 2

Again we inject the error (ignoring inserts of 98):

```
let next_state c s = match c with
  | Push i ->
    if i<>98 then s@[i] else s
    (* remaining cases unchanged *)
```

It is now caught and shrunk as expected:

```
random seed: 388881186
generated error  fail  pass / total     time test name
[✗]    42     0     1    41 / 10000     0.6s queue-model agreement

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

Test queue-model agreement failed (11 shrink steps):

[(Push 60); (Push 98); (Push 8); Pop; Top]
```

Q: Is this a minimal counterexample?

Testing the queues, take 2

Again we inject the error (ignoring inserts of 98):

```
let next_state c s = match c with
  | Push i ->
    if i<>98 then s@[i] else s
    (* remaining cases unchanged *)
```

It is now caught and shrunk as expected:

```
random seed: 388881186
generated error  fail  pass / total     time test name
[✗]    42     0     1    41 / 10000     0.6s queue-model agreement

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

Test queue-model agreement failed (11 shrink steps):

[(Push 60); (Push 98); (Push 8); Pop; Top]
```

Q: Is this a minimal counterexample?

Q: Why do you think the list shrinker failed to cut it further down?

Summary

State-machine models use a functional description of an imperative system to drive QuickCheck tests across random command sequences

Summary

State-machine models use a functional description of an imperative system to drive QuickCheck tests across random command sequences

State-machine models involve:

- 3 types: symbolic commands, model state, and the system under test
- 2 interpreters: over model + system under test
- agreement checking

Summary

State-machine models use a functional description of an imperative system to drive QuickCheck tests across random command sequences

State-machine models involve:

- 3 types: symbolic commands, model state, and the system under test
- 2 interpreters: over model + system under test
- agreement checking

Bonus: we test the interaction between commands

State-dependent command generation may be useful to (a) skew distribution and (b) ensure valid command lists

Some commands come with preconditions that a shrinker has to take into account