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

Jan Midtgaard

MMMI, SDU

SDU

Last week

- State-machine models for testing imperative code
- Dependent generators
- Examples: hashtables and queues

Last lecture's exercises

Today

Today the plan is to cover:

- □ an example of a state machine test in Erlang
- □ ways to test code in other languages
- □ tail calls
- □ fold, map, iter for list processing
- □ generating and shrinking functions
- □ more on properties

Need ports?

You need a QuickCheck port to language X to test software in X, right?

Need ports?

You need a QuickCheck port to language X to test software in X, right?

No

Testing Other Languages 1: FFI (1/2)

From OCaml we can call C functions using a Foreign Function Interface (FFI). Consider, e.g., this C code:

```
int n = 0;     /* a global C variable */

void put(int m)
{ if (n != 538) n = m; } /* an arbitrary injected bug */

int get() { return n; }

void reset() { n = 0; }
```

(example from John Hughes: Certifying your car with Erlang)

Testing Other Languages 1: FFI (1/2)

From OCaml we can call C functions using a Foreign Function Interface (FFI). Consider, e.g., this C code:

```
int n = 0;     /* a global C variable */

void put(int m)
{ if (n != 538) n = m; } /* an arbitrary injected bug */

int get() { return n; }

void reset() { n = 0; }
```

(example from John Hughes: Certifying your car with Erlang)

With the Ctypes library we can describe these as:

```
open Ctypes
open Foreign
let put   = foreign "put" (int @-> returning void)
let get   = foreign "get" (void @-> returning int)
let reset = foreign "reset" (void @-> returning void)
```

SDU

Testing Other Languages 1: FFI (2/2)

A model with QCSTM is now straightforward:

```
module PGConf =
struct
  type cmd = Put of int | Get [@@deriving show { with_path = false }]
  type state = int
  type sut = unit

  let arb_cmd s =
    let int_gen = Gen.oneof [Gen.map Int32.to_int int32.gen; Gen.nat] in
    QCheck.make ~print:show_cmd
      (Gen.oneof [Gen.map (fun i -> Put i) int_gen; Gen.return Get])

  let init_state = 0
  let next_state c s = match c with
    | Put i -> i
    | Get   -> s

  let init_sut () = reset ()
  let cleanup () = ()
  let run_cmd c s () = match c with
    | Put i -> begin put i; true end
    | Get   -> (get () = s)

  let precond _ _ = true
end
```

SDU

FFI pros and cons

In many ways this works well:

- We write type-safe testing code
- A high-level language for low-level language testing to clearly capture a specification

FFI pros and cons

In many ways this works well:

- We write type-safe testing code
- A high-level language for low-level language testing to clearly capture a specification

There are also disadvantages though:

- The C code could potentially/likely get into an erroneous state requiring a reload to recover
- If the C code crashes it also takes the tester down

FFI pros and cons

In many ways this works well:

- We write type-safe testing code
- A high-level language for low-level language testing to clearly capture a specification

There are also disadvantages though:

- The C code could potentially/likely get into an erroneous state requiring a reload to recover
- If the C code crashes it also takes the tester down

Dynamic linking can solve the first issue (put-get example from `https://github.com/jmid/qcstm` does this, non-Windows I believe)

The second issue requires running separate processes

Testing Other Languages 2: Compiling (1/4)

Reconsider our state machine model:

```
module PGConf =
struct
  type cmd = Put of int | Get [@@deriving show { with_path = false }]
  type state = int
  type sut = unit

  let arb_cmd s =
    let int_gen = Gen.oneof [Gen.map Int32.to_int int32.gen; Gen.nat] in
    QCheck.make ~print:show_cmd
      (Gen.oneof [Gen.map (fun i -> Put i) int_gen; Gen.return Get])

  let init_state = 0
  let next_state c s = match c with
    | Put i -> i
    | Get   -> s

  let init_sut () = reset ()
  let cleanup () = ()
  let run_cmd c s () = match c with
    | Put i -> begin put i; true end
    | Get   -> (get () = s)

  let precond _ _ = true
end
```

Testing Other Languages 2: Compiling (1/4)

Reconsider our state machine model:

```
module PGConf =
struct
  type cmd = Put of int | Get [@@deriving show { with_path = false }]
  type state = int
  type sut = unit

  let arb_cmd s =
    let int_gen = Gen.oneof [Gen.map Int32.to_int int32.gen; Gen.nat] in
    QCheck.make ~print:show_cmd
      (Gen.oneof [Gen.map (fun i -> Put i) int_gen; Gen.return Get])

  let init_state = 0
  let next_state c s = match c with
    | Put i -> i
    | Get   -> s

  let init_sut () = reset ()
  let cleanup () = ()
  let run_cmd c s () = match c with
    | Put i -> begin printf " put(%i);\n" i; true end
    | Get   -> begin printf " assert(get () == %i);\n" s; true end

  let precond _ _ = true
end
```

Suppose we output C code instead of calling it?

Testing Other Languages 2: Compiling (2/4)

In effect our interpreter over command lists

- no longer interprets each command over the FFI
- but instead emits code:

```
# PGtest.interp_agree 0 () (Gen.generate1 (PGtest.arb_cmds 0).gen);;
 assert(get () == 0);
 put(0);
 assert(get () == 0);
 put(1848846579);
 assert(get () == 1848846579);
 put(1710249865);
 assert(get () == 1710249865);
- : bool = true
#
```

Testing Other Languages 2: Compiling (2/4)

In effect our interpreter over command lists

- □ no longer interprets each command over the FFI
- □ but instead emits code:

```
# PGtest.interp_agree 0 () (Gen.generate1 (PGtest.arb_cmds 0).gen);;
 assert(get () == 0);
 put(0);
 assert(get () == 0);
 put(1848846579);
 assert(get () == 1848846579);
 put(1710249865);
 assert(get () == 1710249865);
- : bool = true
#
```

We just need to

- □ write the code to a file instead
- □ add a prefix **`#include ...`** and suffix `return 0;`
- □ compile and run it

Testing Other Languages 2: Compiling (3/4)

We change the `sut` to be an output stream:

```
module PGConf =
struct
  (* ... *)
  type sut = out_channel (* was: unit *)
  (* ... *)

  let init_sut () =
    let ostr = open_out "tmp.c" in
    begin
      fprintf ostr "#include␣<assert.h>\n";
      fprintf ostr "int␣main()␣{\n";
      ostr
    end
  let cleanup ostr =
    begin
      fprintf ostr "␣return␣0;\n";
      fprintf ostr "}\n";
      flush ostr;
      close_out ostr
    end
  let run_cmd c s ostr = match c with
    | Put i -> begin fprintf ostr "␣put(%i);\n" i; true end
    | Get   -> begin fprintf ostr "␣assert(get␣()␣==␣%i);\n" s; true end
  (* ... *)
end
```

Testing Other Languages 2: Compiling (4/4)

Finally we can put it all together:

```
Test.make ~name:"compiled␣putget" ~count:500
  (PGtest.arb_cmds PGConf.init_state) (* generator of commands *)
  (fun cs ->
     let ostr = PGConf.init_sut () in
     ignore(PGtest.interp_agree PGConf.init_state ostr cs);
     PGConf.cleanup ostr;
     (* now compile and run program, checking exit codes *)
     0 = Sys.command ("gcc␣-Wall␣...␣putgetlib.c␣tmp.c␣-o␣tmp")
     && 0 = Sys.command ("exec␣2>tmp.stderr;␣./tmp␣1>tmp.stdout"))
```

Testing Other Languages 2: Compiling (4/4)

Finally we can put it all together:

```
Test.make ~name:"compiled␣putget" ~count:500
  (PGtest.arb_cmds PGConf.init_state) (* generator of commands *)
  (fun cs ->
     let ostr = PGConf.init_sut () in
     ignore(PGtest.interp_agree PGConf.init_state ostr cs);
     PGConf.cleanup ostr;
     (* now compile and run program, checking exit codes *)
     0 = Sys.command ("gcc␣-Wall␣...␣putgetlib.c␣tmp.c␣-o␣tmp")
     && 0 = Sys.command ("exec␣2>tmp.stderr;␣./tmp␣1>tmp.stdout"))
```

This works remarkably well!

```
generated error fail pass / total     time test name
[✗]    31     0    1   30 /  500    10.3s compiled putget

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

Test compiled putget failed (12 shrink steps):

[(Put 538); (Put -1107714141); Get]
```

It is slower than the FFI though, due to context switching and reading+writing to disk

Tail Calls

Functional programming: Non-tail calls

Consider the following recursive function for adding the elements of an integer list:

```
let rec sum xs = match xs with
  | [] -> 0
  | x::xs -> x + sum xs
```

It requires in the order of $|\texttt{xs}|$ stack frames on the call stack.

This does not scale to big lists:

```
# Gen.(generate1 (list_size (return 10) nat));;
- : int list = [6; 5; 1; 4; 648; 2; 2; 603; 534; 515]
# sum Gen.(generate1 (list_size (return 200) nat));;
- : int = 98405
# sum Gen.(generate1 (list_size (return 200000) nat));;
Stack overflow during evaluation (looping recursion?).
#
```

Functional programming: Tail calls

This variant instead accumulates the sum in `acc`:

```
let sum' xs =
  let rec sum_local xs acc = match xs with
    | [] -> acc
    | x::xs -> sum_local xs (x+acc)
  in sum_local xs 0
```

Functional programming: Tail calls

This variant instead accumulates the sum in `acc`:

```
let sum' xs =
  let rec sum_local xs acc = match xs with
    | [] -> acc
    | x::xs -> sum_local xs (x+acc)
  in sum_local xs 0
```

It requires only constant stack space!

```
# sum' Gen.(generate1 (list_size (return 500000) nat));;
- : int = 183297676
```

Functional programming: Tail calls

This variant instead accumulates the sum in `acc`:

```
let sum' xs =
  let rec sum_local xs acc = match xs with
    | [] -> acc
    | x::xs -> sum_local xs (x+acc)
  in sum_local xs 0
```

It requires only constant stack space!

```
# sum' Gen.(generate1 (list_size (return 500000) nat));;
- : int = 183297676
```

Why? The result of the recursive call is also the result of the non-empty branch. No need to return-to-return – so let's not push a call stack frame!

Functional programming: Tail calls

This variant instead accumulates the sum in `acc`:

```
let sum' xs =
  let rec sum_local xs acc = match xs with
    | [] -> acc
    | x::xs -> sum_local xs (x+acc)
  in sum_local xs 0
```

It requires only constant stack space!

```
# sum' Gen.(generate1 (list_size (return 500000) nat));;
- : int = 183297676
```

Why? The result of the recursive call is also the result of the non-empty branch. No need to return-to-return – so let's not push a call stack frame!

Such *"last calls"* are called *tail calls*. The optimization is called *tail-call optimization*. It turns recursion into a loop!

Most functional languages (+ Lua) perform it.

Functional programming: List folding

Functional programming often involves list traversal:

```
let rec sum xs = match xs with
  | [] ->  0
  | x::xs -> x + sum xs

let rec concat xs = match xs with
  | [] -> ""
  | x::xs -> x ^ concat xs
```

This is almost identical code! What differs is:

Functional programming: List folding

Functional programming often involves list traversal:

```
let rec sum xs = match xs with
  | [] ->  0
  | x::xs -> x + sum xs

let rec concat xs = match xs with
  | [] -> ""
  | x::xs -> x ^ concat xs
```

This is almost identical code! What differs is:

- the base case (for empty lists)

Functional programming: List folding

Functional programming often involves list traversal:

```
let rec sum xs = match xs with
  | [] ->  0
  | x::xs -> x + sum xs

let rec concat xs = match xs with
  | [] -> ""
  | x::xs -> x ^ concat xs
```

This is almost identical code! What differs is:

- the base case (for empty lists)
- how to compose the result for a non-empty list

Functional programming: List folding

Functional programming often involves list traversal:

```
let rec sum xs = match xs with
  | [] ->  0
  | x::xs -> x + sum xs

let rec concat xs = match xs with
  | [] -> ""
  | x::xs -> x ^ concat xs
```

This is almost identical code! What differs is:

- the base case (for empty lists)
- how to compose the result for a non-empty list

Within FP this is concisely expressed with a *fold*:

```
let sum xs = List.fold_left (fun acc x -> acc + x) 0 xs
let concat xs = List.fold_left (fun acc s -> acc ^ s) "" xs
```

Functional programming: List folding

Functional programming often involves list traversal:

```
let rec sum xs = match xs with
  | [] ->  0
  | x::xs -> x + sum xs

let rec concat xs = match xs with
  | [] -> ""
  | x::xs -> x ^ concat xs
```

This is almost identical code! What differs is:

- the base case (for empty lists)
- how to compose the result for a non-empty list

Within FP this is concisely expressed with a *fold*:

```
let sum xs = List.fold_left (+) 0 xs
let concat xs = List.fold_left (^) "" xs
```

. . . with infix operator syntax

Functional programming: List folding

Functional programming often involves list traversal:

```
let rec sum xs = match xs with
  | [] ->  0
  | x::xs -> x + sum xs

let rec concat xs = match xs with
  | [] -> ""
  | x::xs -> x ^ concat xs
```

This is almost identical code! What differs is:

- the base case (for empty lists)
- how to compose the result for a non-empty list

Within FP this is concisely expressed with a *fold*:

```
let sum = List.fold_left (+) 0
let concat = List.fold_left (^) ""
```

. . . with infix operator syntax . . . and eta reduction

Other common list functions: `map` and `iter`

`fold` is typically used for combining list elements

`map` instead performs an operation on each element individually:

```
# let double x = x+x;;
val double : int -> int = <fun>
# List.map double [1;2;3;4];;
- : int list = [2; 4; 6; 8]
```

Other common list functions: `map` and `iter`

`fold` is typically used for combining list elements

`map` instead performs an operation on each element individually:

```
# let double x = x+x;;
val double : int -> int = <fun>
# List.map double [1;2;3;4];;
- : int list = [2; 4; 6; 8]
```

`iter` also performs an operation on each element – but for its side-effect:

```
# List.iter (fun i -> Printf.printf "%i␣" i) [1;2;3;4];;
1 2 3 4 - : unit = ()
```

Compare their type signatures:

```
val map : ('a -> 'b) -> 'a list -> 'b list
val iter : ('a -> unit) -> 'a list -> unit
```

Testing `fold`, `map`, . . . (1/3)

Since `fold`, `map`, . . . take functions as input,
we need to generate arbitrary functions to test them!

With QCheck we can actually do that!

Caveat: it only generates pure functions

The key insight is that a test can only observe a function at a finite number of arguments.

For example, `fun1 Observable.int small_string` creates a full generator of `int -> string` functions.
The full generator type of this example is:

```
(int -> string) fun_ arbitrary
```

whereas the underlying pure generator type is:

SDU

```
(int -> string) fun_ Gen.t
```

Testing `fold`, `map`, ... (2/3)

QCheck also has tools for building function generators:

- an `Observable` module for argument types:
  - with base types: `bool`, `char`, `float`, ...
  - with combinators: `pair`, `triple`, ...
- `fun2`, `fun3`, `fun4` for multi-argument functions

As you can tell from the type:
`(int -> string) fun_ arbitrary` function generators are not internally represented with functions

To apply a generated function, we need to coerce it from the internal representation with `Fn.apply`:

```
Fn.apply : 'f fun_ -> 'f
```

Testing `fold`, `map`, ... (3/3)

To test `List.fold_left` we need concrete argument and result types. An example:

```
Test.make
  (quad  (* string -> int -> string *)
     (fun2 Observable.string Observable.int small_string)
     small_string
     (list small_nat)
     (list small_nat))
  (fun (f,acc,is,js) ->
     let f = Fn.apply f in
     List.fold_left f acc (is @ js)
       = List.fold_left f (List.fold_left f acc is) js)
```

Here I test folding over `int list`s with

- a `string` accumulator and
- an arbitrary `string -> int -> string` function

Shrinking functions (1/2)

Function generators also have shrinkers.
Beware: shrinking order starts to matter:

```
Test.make ~name:"false␣fold,␣fun␣first"
  (quad  (* string -> int -> string *)
     (fun2 Observable.string Observable.int small_string)
     small_string
     (list small_nat)
     (list small_nat))
  (fun (f,acc,is,js) ->
     let f = Fn.apply f in
     List.fold_left f acc (is @ js)
       = List.fold_left f (List.fold_left f acc is) is)
```

With this typo the property is false, but it takes 2 sec – 4+ min to shrink to a minimal counterexample:

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

Test false fold, fun first failed (41 shrink steps):

({(4) -> "b"; _ -> ""}, "", [], [4])
```

Shrinking functions (2/2)

By simply rearranging the tuple, it consistently takes 0.0 sec to find and shrink to a minimal counterexample:

```
Test.make ~name:"false␣fold,␣lists␣first"
  (quad  (* string -> int -> string *)
     (list small_nat)
     (list small_nat)
     (fun2 Observable.string Observable.int small_string)
     small_string)
  (fun (is,js,f,acc) ->
     let f = Fn.apply f in
     List.fold_left f acc (is @ js)
       = List.fold_left f (List.fold_left f acc is) is)
```

(this is with a different gen. order, hence different seed)

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

Test false fold, lists first failed (25 shrink steps):

([], [0], {_ -> ""}, "\230")
```

SDU

Why? the `quad` shrinker tries left-to-right. . .

More on Properties

Properties, generally

What properties should one test for?

- In an unsafe language a first property could simply be *“doesn’t crash”*.

  In C/C++/. . . code this can find actual errors

- For a stateful system, agreement with a state-machine model is a natural suggestion.

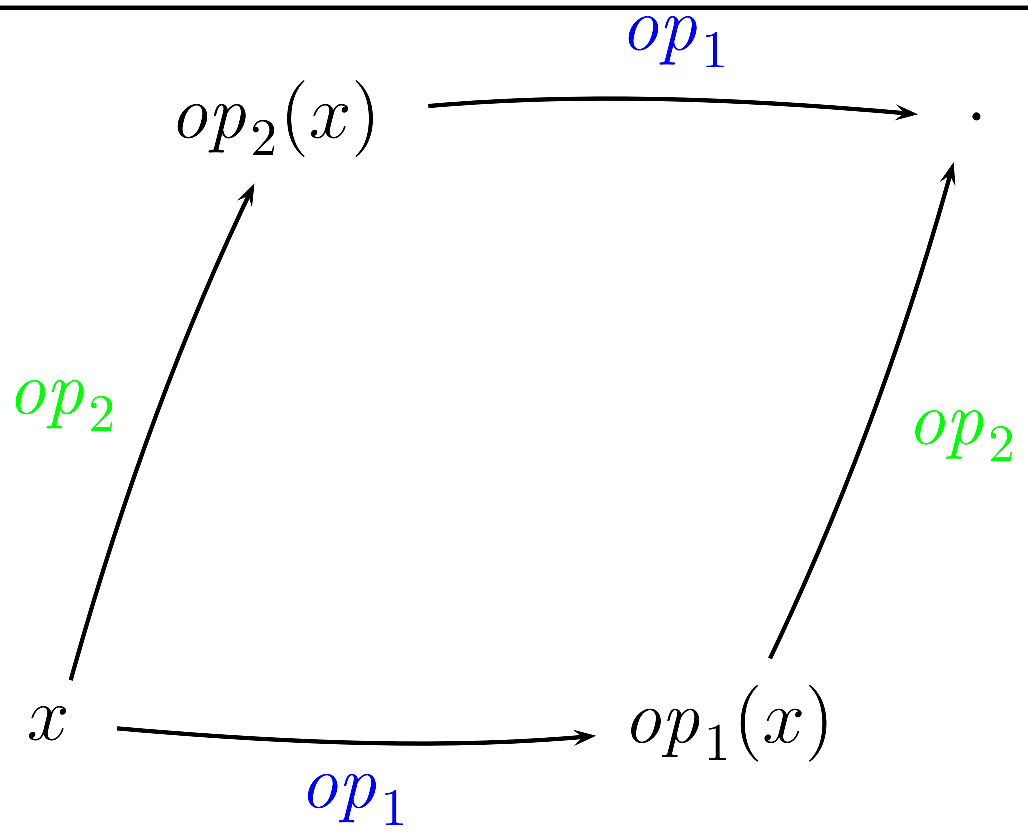
- Sometimes you have an oracle which you can test against.

  Example: testing an advanced data structure against a simpler, naive implementation (Patricia trees)

These are general property guidelines

There are other general patterns (Scott Wlaschin)

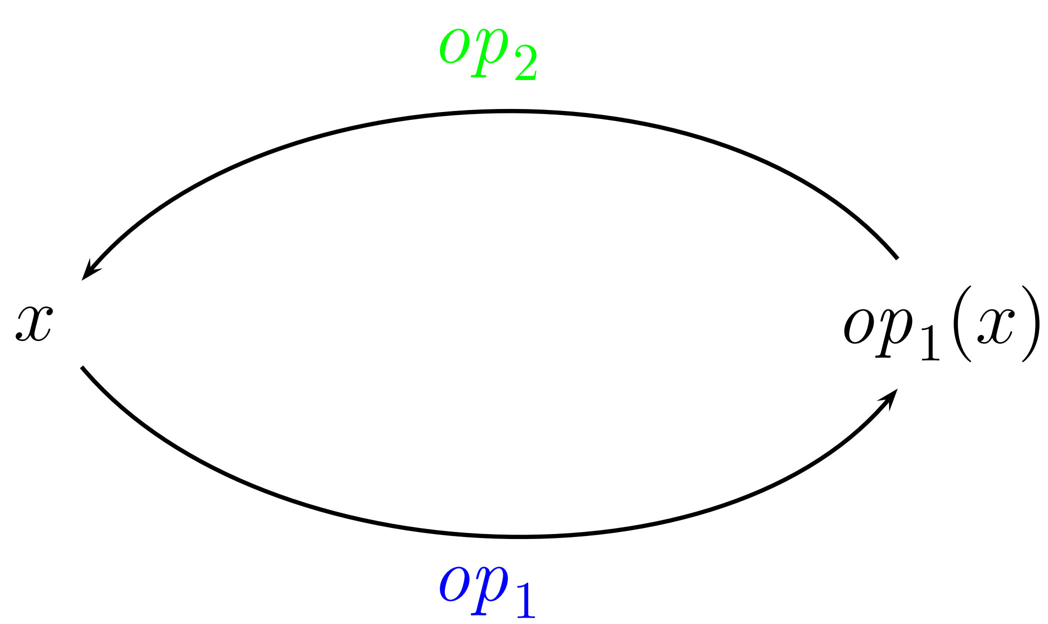
Commuting diagram (“different paths, same destination”)



A common property is that two different sequences of operations should yield the same result.

Examples:
model-impl. agreement in model-based approach,
rev-concat vs concat-rev, interp. vs compile-run, . . .

Inverses ("there and back again")



Another common property is that two operations act as inverses

Examples:

`Int64.to_int+Int64.of_int`, encryption+decryption, prettyprint-parse, add-subtract, exp-log, reverse-reverse, serialize-deserialize, insert-remove, add-lookup

Related inputs lead to related outputs

Another common (relational) property is that an operation on two related inputs should give rise to two related outputs

Examples:

- Congruence $i \sim i' \implies f(i) \sim f(i')$
  HTTP requests w/sim. headers give sim. responses, equivalent sets/data structures repr. differently in memory produce equivalent results
- Monotonicity/anti-tonicity $i \leq i' \implies f(i) \leq f(i')$
  (string search, shortests path, data-flow analysis, . . .)
  In general, "bigger input" should lead to "bigger result" (for suitable ordering, e.g., interpreting **false** $\leq$ **true** for `member`)

Invariants (“some things never change”)

Common to many data structures (but also many programs) is an invariant

(something that doesn’t change or vary)

Examples:

- red-black invariant,
- search-tree invariant,
- sorting preserves length,
- sorting preserves elements,
- “counter represents number of elements in data-structure or database”
- . . .

Idempotency (“The more things change, the more they stay the same”)

Another common property is that several invocations of the same operation does not change the outcome.

Examples:

- sorting $\texttt{sort}\ l = \texttt{sort}\ (\texttt{sort}\ l)$,
- `String.lowercase`, `String.uppercase`,
- Idempotent HTTP requests (GET, PUT, DELETE, . . .)

  https://tools.ietf.org/html/rfc7231#section-4.2.2
- . . .

In imperative code, left-over internal state sometimes leads two calls with same input to return two different results. . .

Structural induction ("Solve a smaller problem first")

Some properties lend themselves to be broken up into a property for a sub-problem, akin to how we prove a property using structural induction.

Example:

- Sorting: a list is sorted if it has
  - zero or one element (base cases)
  - two or more elements, the first two are sorted, and the list's tail is sorted (inductive hypothesis)

```
let rec sorted xs = match xs with
  | []  -> true
  | [x] -> true
  | x::y::xs' -> x <= y && sorted (y::xs')
```

This expresses sortedness in terms of a sorted tail.

Easier to verify ("hard to prove, easy to verify")

A number of problems in CS are hard to solve,
but much easier to check.

Examples:

- prime-number factorization,
- sorting vs. check sorted,
- path finding,
- tokenization,
- any NP-complete problem
  (SAT, traveling salesman, graph colouring, . . .),
- fixed-point computation vs. checking,
- . . .

Blackbox or whitebox properties?

Property-based testing is not limited to either whitebox or blackbox properties:

- The red-black trees is an example of a data structure invariant — a whitebox property
- The model-based approach in the Patricia tree example, and the `Queue` and `Hashtable` state machine examples are **blackbox properties**

  “How should the individual API operations interact / operate abstractly?”

Improving properties with devil's advocate

Sometimes you have half an idea for a relevant property

In such a situation
it can be useful to play devil's advocate:

"Which erroneous implementation
could escape these tests?"

Improving properties with devil's advocate

Sometimes you have half an idea for a relevant property

In such a situation
it can be useful to play devil's advocate:

"Which erroneous implementation
could escape these tests?"

"How can I patch my properties such that it doesn't?"

Improving properties with devil's advocate

Sometimes you have half an idea for a relevant property

In such a situation
it can be useful to play devil's advocate:

"Which erroneous implementation
could escape these tests?"

"How can I patch my properties such that it doesn't?"

Example:

```
(fun xs -> xs = List.rev (List.rev xs))
```

Q: Which implementation can fly under this radar?

Improving properties with devil's advocate

Sometimes you have half an idea for a relevant property

In such a situation
it can be useful to play devil's advocate:

"Which erroneous implementation
could escape these tests?"

"How can I patch my properties such that it doesn't?"

Example:

```
(fun xs -> xs = List.rev (List.rev xs))
```

Q: Which implementation can fly under this radar?

Q: How can we add a property to catch it?

Summary

Today we've talked about

- two ways (FFI + emitting code) to test code in other languages
- tail calls (turning recursive function into loops)
- fold, map, iter for typical list processing
- QCheck's ability to generate and shrink functions
- general ideas for properties and ways to strengthen a property

+ We have seen an example of a model-based state machine test in Erlang