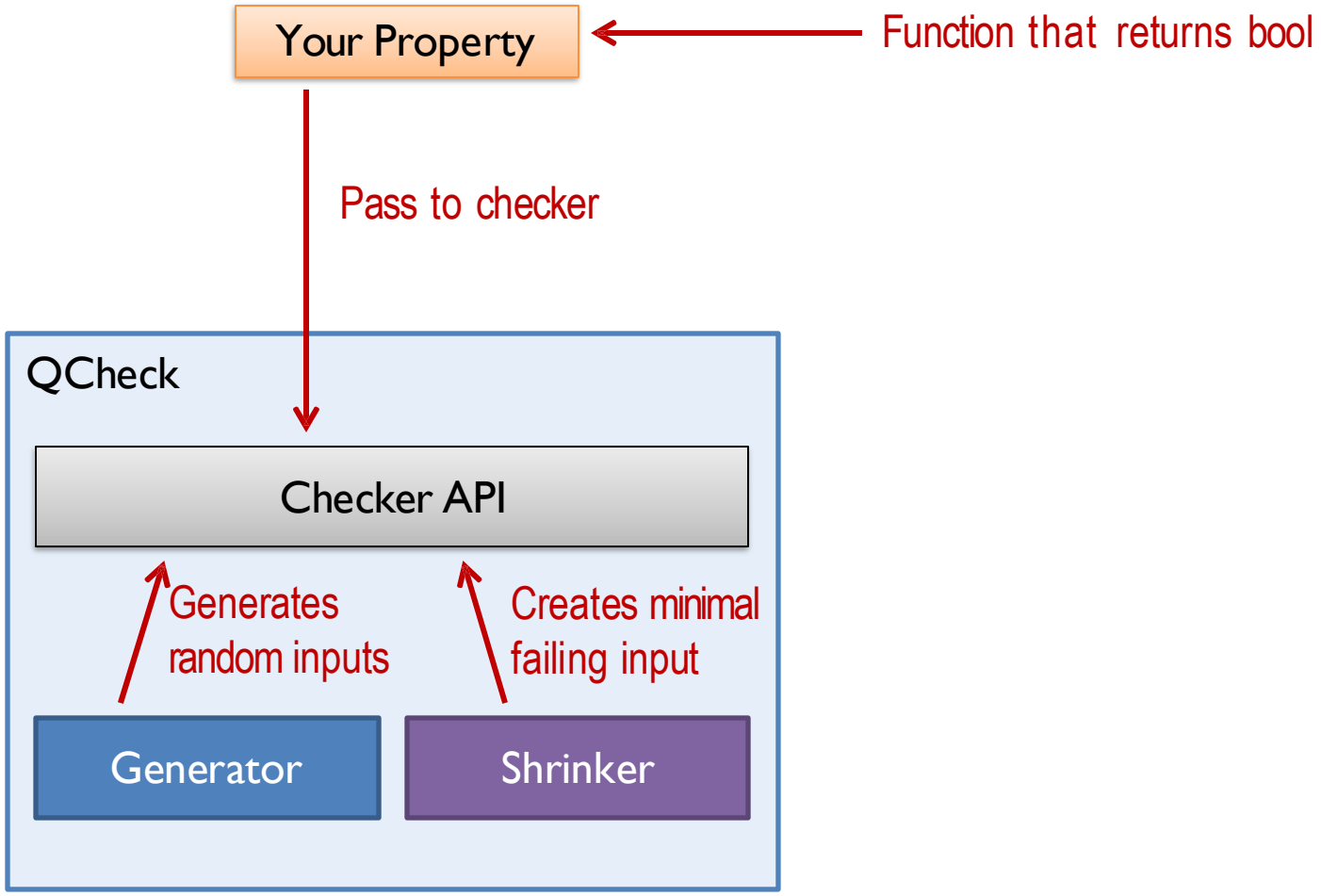
From Properties to Tests

COS 326

David Walker

QCheck Overview



Using QCheck

Write some properties: t -> bool

```
type iprop = int * int -> bool

let commutes f (x,y) =
  f x y = f y x
let add_commutes p =
  commutes (+) p
let sub_commutes p =
  commutes (-) p
```

Create a generator

```
module Q = Qcheck

type 'a gen = 'a Q.arbitrary

let ints = Q.int
let pairs = Q.tup2 ints ints
```

Create a test

```
let t1 =
  Q.Test.make
       ~name:"add commutes"
       ~count:100
       pairs
       add_commutes
```

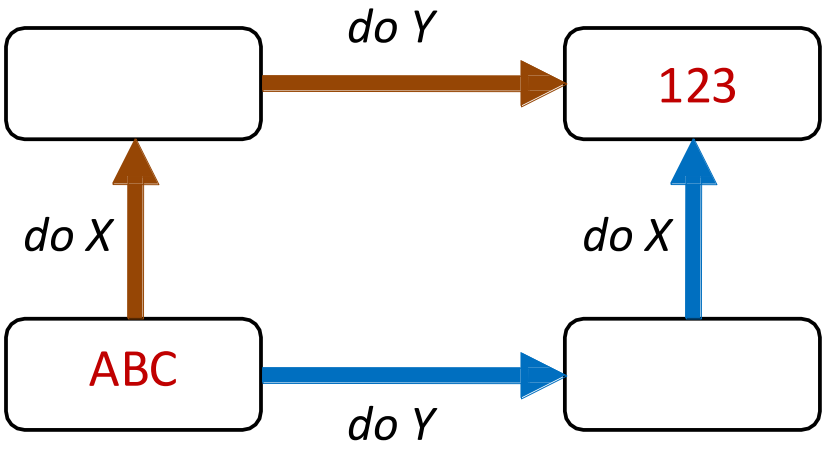
```
let _ =
  QCheck_base_runner.run_tests
    ~verbose:true
    [t1;t2]
```

Example-based tests vs. Property-based tests

- PBTs are more general
  - One property-based test can replace many example-based tests.
- PBTs can reveal overlooked edge cases
  - Nulls, negative numbers, weird strings, etc.
- PBTs ensure deep understanding of requirements
  - Property-based tests force you to think! ☹
- PBTs can do shrinking to find the boundary cases!
- Example-based tests are still helpful though!
  - Less abstract, easier to understand

Techniques for Creating Specifications

"Different paths, same destination"

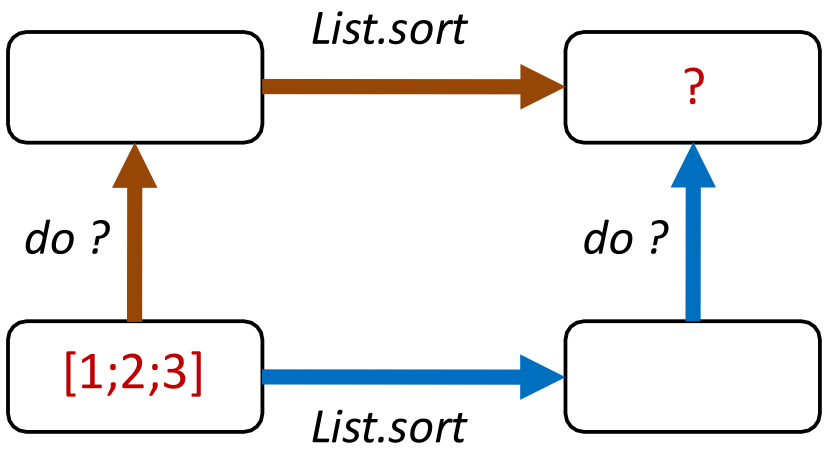


Examples:

- Commutivity
- Associativity
- Map f then Map g

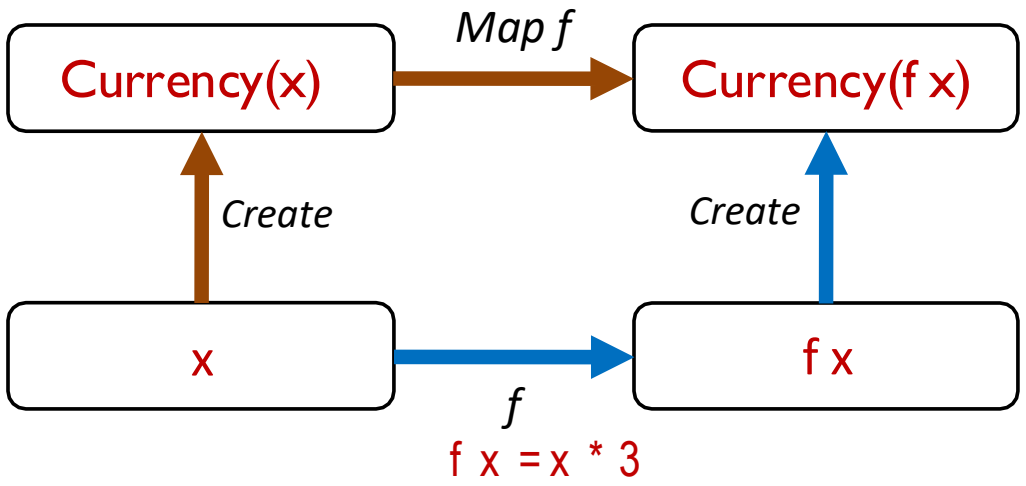
"Different paths, same destination"

Applied to a sort function



"Different paths, same destination"

Applied to a map function

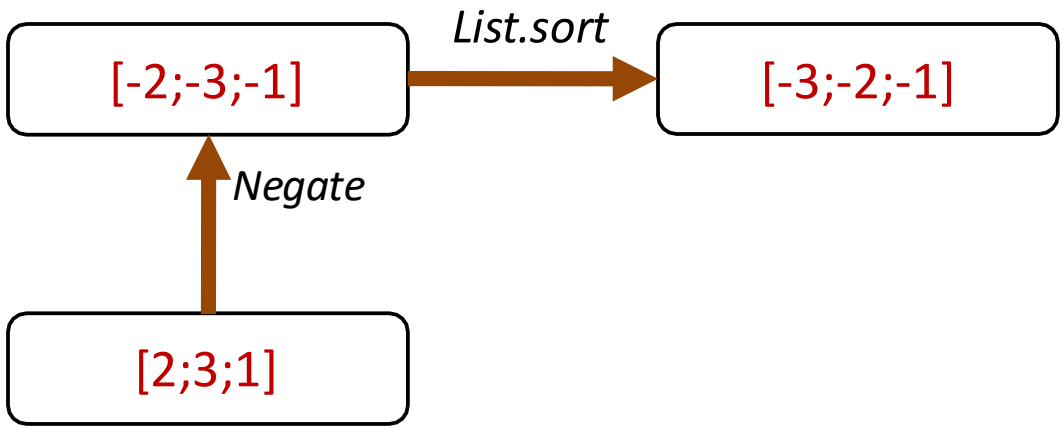



```
Currency(2)
  .Map(x => x * 3)
```

```
Currency(2 * 3)
```

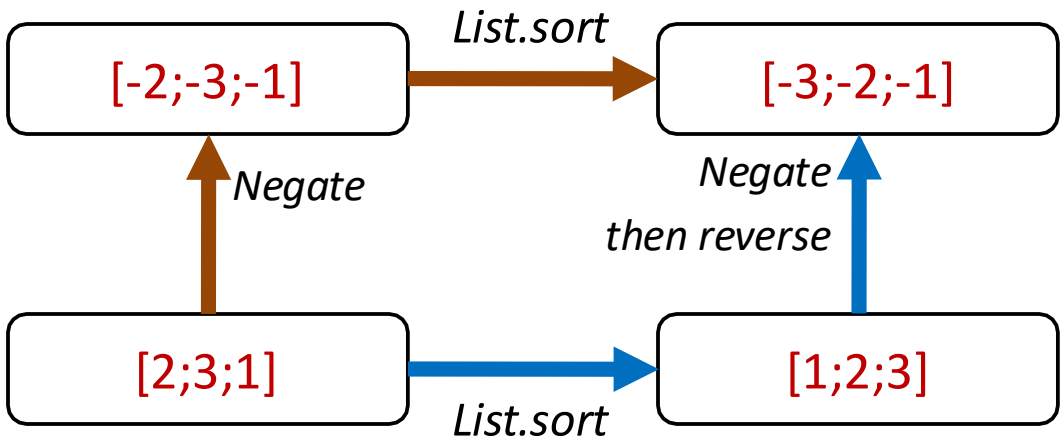
"Different paths, same destination"

Applied to a sort function

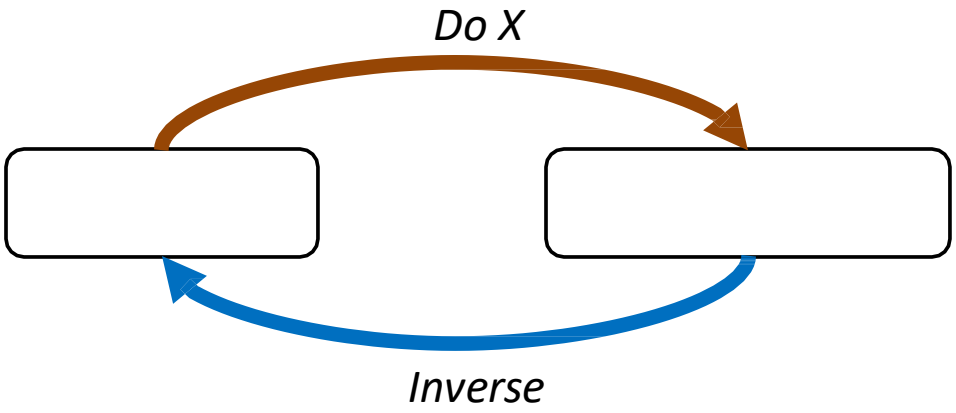


"Different paths, same destination"

Applied to a sort function



"There and back again"

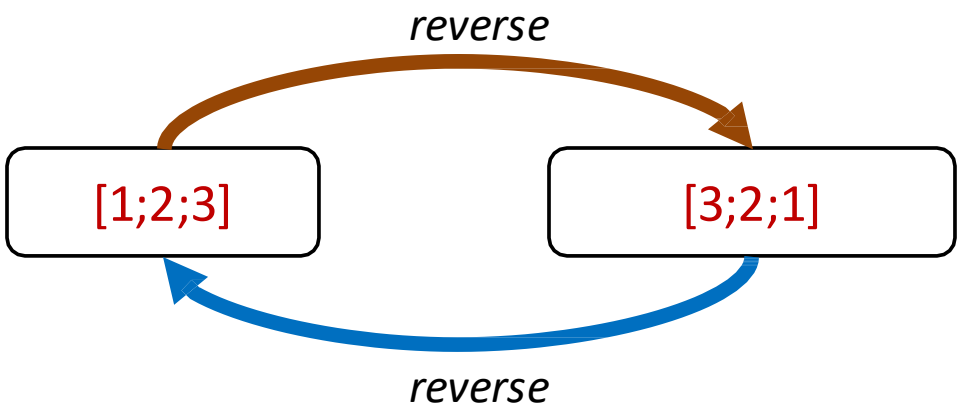


Examples:

- Serialization/Deserialization
- Addition/Subtraction
- Write/Read
- SetProperty/GetProperty

"There and back again"

Applied to a list reverse function



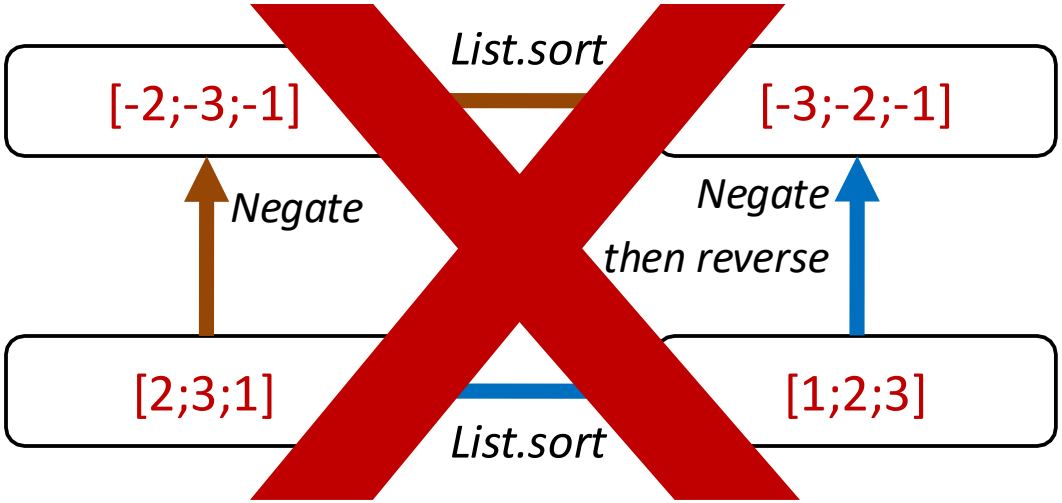
Pro tip:
We often need a combination of properties, not just one

We needed three properties
to define "add"

"Some things never change"

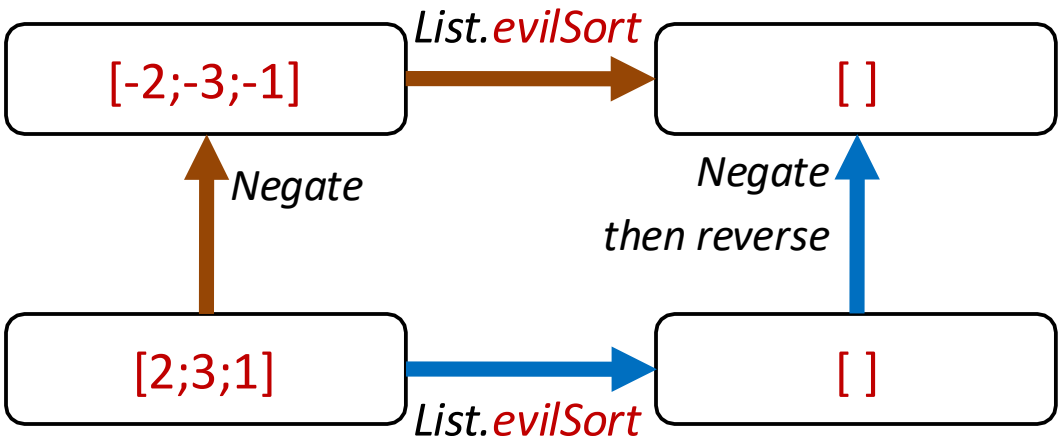
Examples:
- Size of a collection
- Contents of a collection
- Balanced trees
  (balanced before implies balanced after any operation)

The EDFH and List.Sort



The EDFH can beat this!

The EDFH and List.Sort

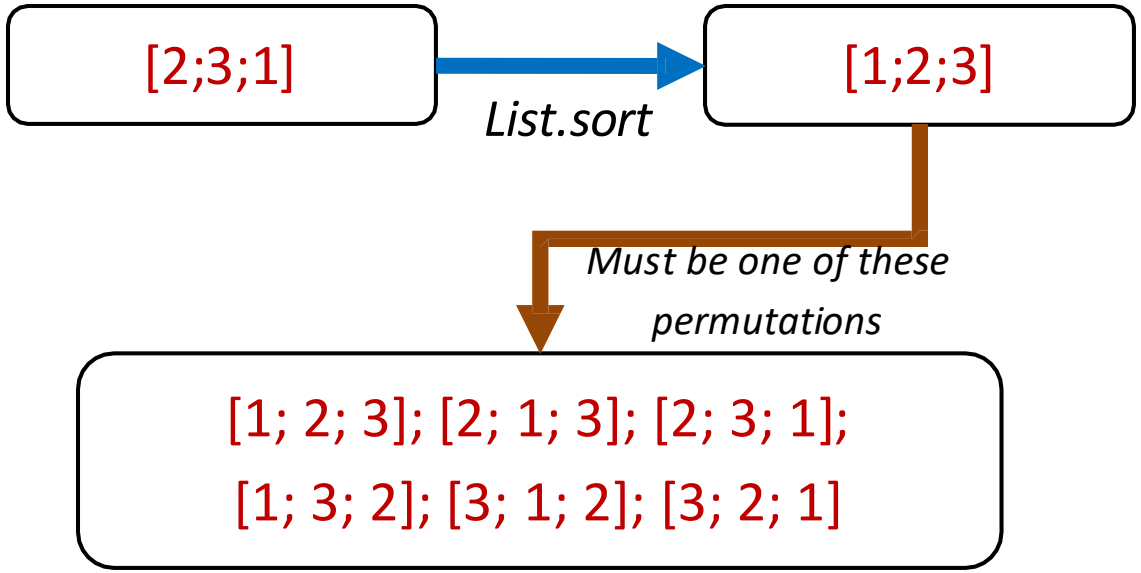


EvilSort just returns an empty list!

This passes the "commutivity" test!

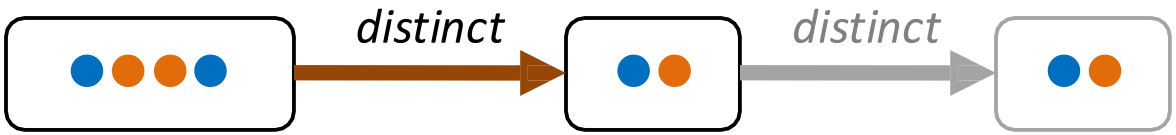
"Some things never change"

Used to ensure the sort function is good



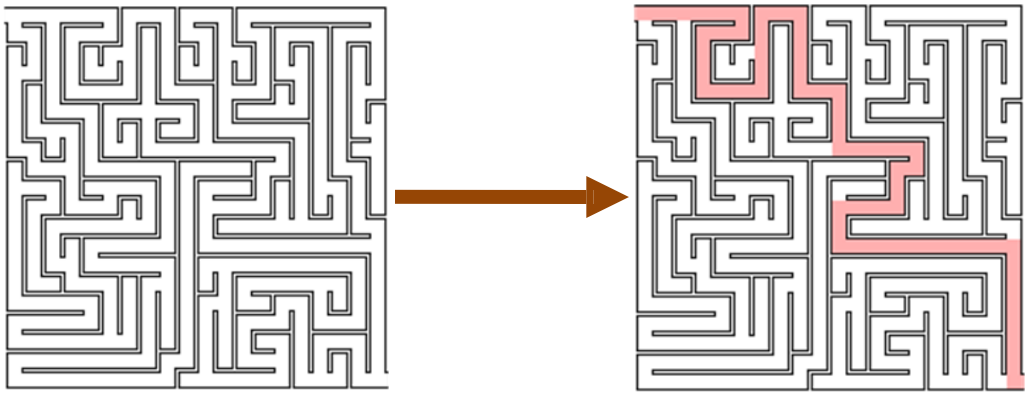
The EDFH is beaten now!

"The more things change, the more they stay the same"



Idempotence:
- Sort
- Filter
- Event processing
- Required for distributed designs

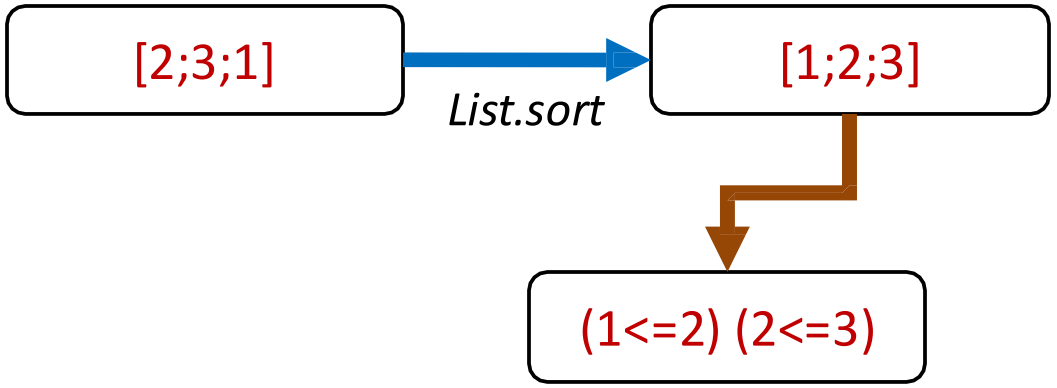
"Hard to prove, easy to verify"



- Prime number factorization
- Too many others to mention!

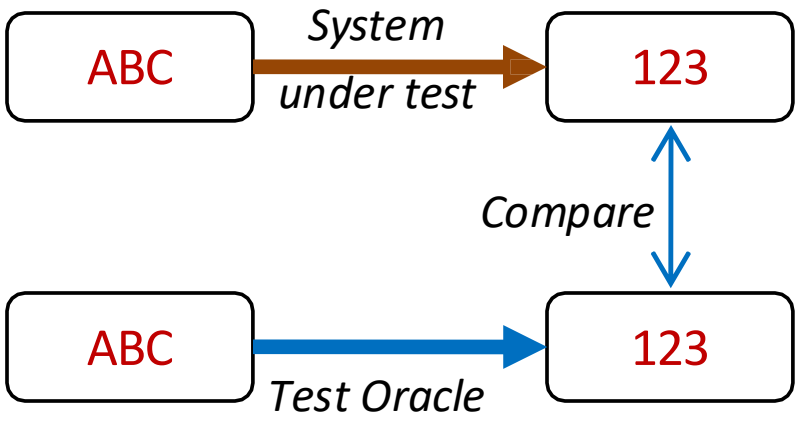
"Hard to prove, easy to verify"

Applied to a sort



To verify the sort,
check that each pair is ordered

"The test oracle"

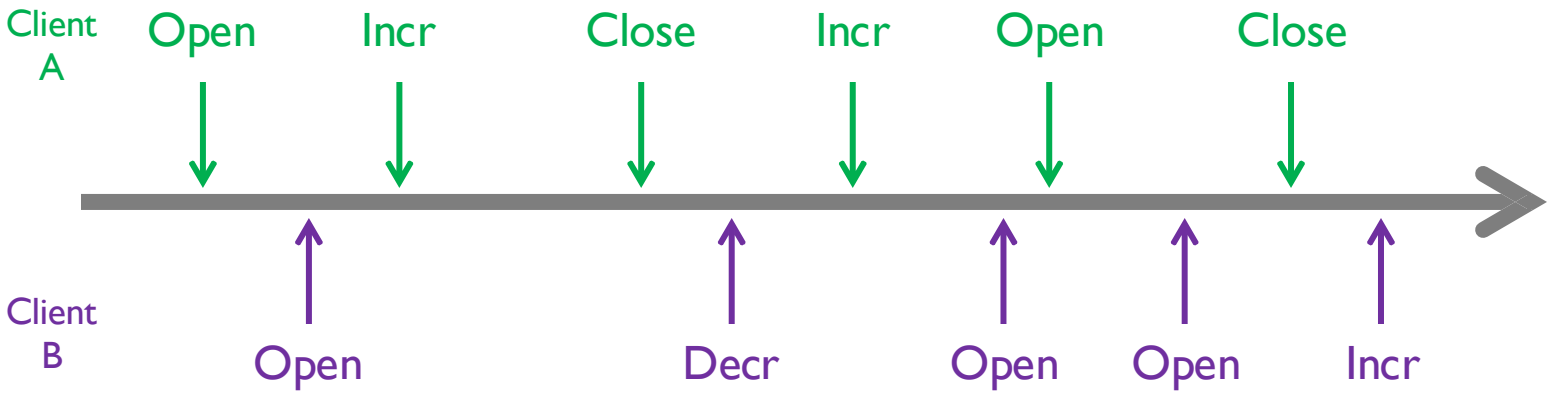


- Compare optimized with slow brute-force version
- Compare parallel with single thread version
- Legacy system is the oracle for a replacement system

Testing a simple database

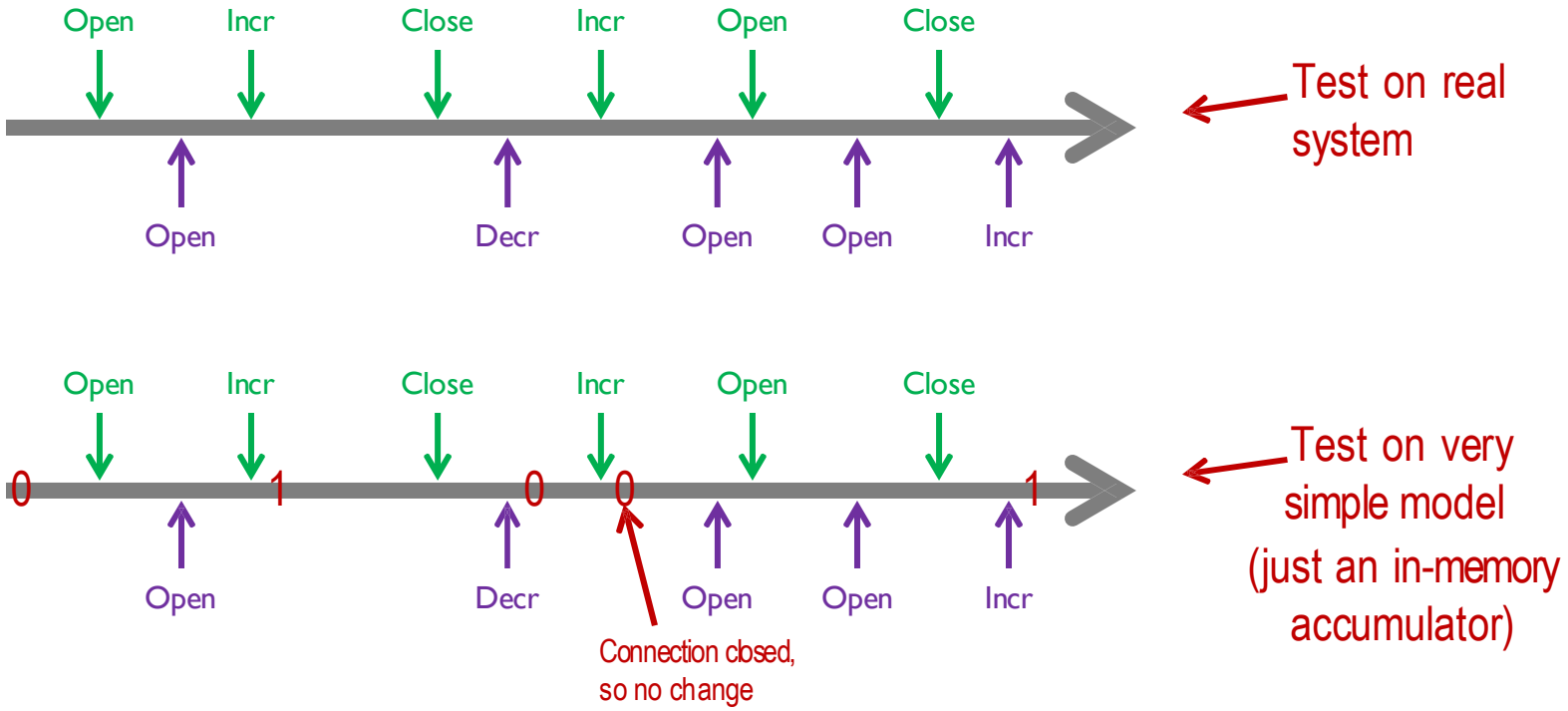
Four operations: Open, Close, Increment, Decrement
Two clients: Client A and Client B

Let QuickCheck generate a random list of these actions for each client



How do use this to check that our db works?

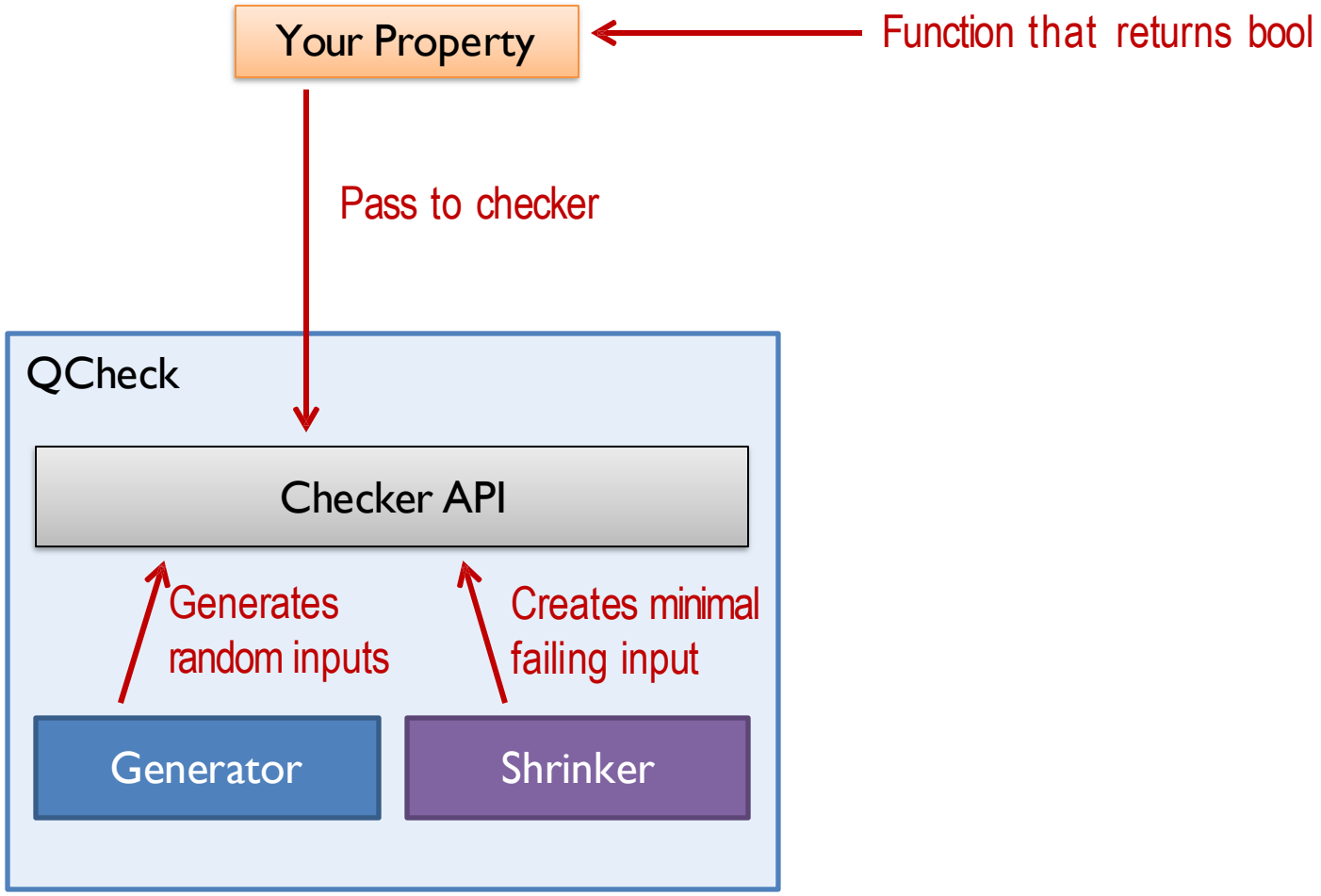
Testing a simple database



Compare model result with real system!

Creating Generators

QCheck Overview



Using QCheck

Write some properties: t -> bool

```
let commutes f (x,y) =
  f x y = f y x

let add_commutes p =
  commutes (+) p
let sub_commutes p =
  commutes (-) p
```

Create a generator

```
module Q = Qcheck

type 'a arb = 'a Q.arbitrary

let ints = Q.int
let pairs = Q.tup2 ints ints
```

Create tests

```
let t1 =
  Q.Test.make
       ~name:"add commutes"
       ~count:100
       pairs
       add_commutes
```

Run tests

```
let _ =
  QCheck_base_runner.run_tests
    ~verbose:true
    [t1;t2]
```

Browsing Interfaces with utop

in your terminal:

```
% opam install qcheck
```

```
% utop
```

https://github.com/c-cube/qcheck/blob/main/src/core/QCheck.mli

in utop:

```
utop # #require "qcheck";;
utop # #show QCheck;;
```

← name of “package”

← name of module

```
module QCheck :
  sig
    val ( ==> ) : bool -> bool -> bool
    val assume : bool -> unit
    val assume_fail : unit -> 'a
    module Gen : sig ... end
    module Print : sig ... end
    …
```

results of #show Qcheck

(and many more lines)

Creating Generators

Create a generator

```
type 'a arb = 'a Q.arbitrary

let ints : int arb = Q.int
let pairs (int * int) arb = Q.tup2 ints ints
```

But what if we are dealing with our own data types?

```
type color = Red | Green | Blue

type 'a tree = Leaf of 'a | Tree of 'a tree * 'a tree
```

Or if we want to control the distribution?
Eg: generate ints, but be sure we generate a lot of powers of 2?

The ‘a arbitrary type

Create a generator

```
type 'a arb = 'a Q.arbitrary

let ints : int arb = Q.int
let pairs (int * int) arb = Q.tup2 ints ints
```

What is in an “arbitrary”? A data structure containing:

- an optional print function
- an optional shrinker object
- some other things
- a generator object with type **‘a Gen.t**

Let’s focus on the generator object

```
Q.make : … optional arguments -> 'a Gen.t -> 'a arb
```

The ‘a Gen.t type

```
utop # #show Qcheck.Gen;;
module Gen :
  sig
    type 'a t = Random.State.t -> ‘a
    …
```

The main part of an arbitrary is a generator.

A generator is a function from “random state” to value.

If we want to make new generators, we ultimately have to build new functions

The ‘a Gen.t type

```
utop # #show Qcheck.Gen;;
module Gen :
  sig
    type 'a t = Random.State.t -> ‘a
    …
```

A generator is a function from “random state” to value.

If we want to make new generators, we ultimately have to build new functions

the Gen module gives lots of help doing that

The ‘a Gen.t module

```
val unit : unit t      (** The unit generator. *)

val bool : bool t.     (** The boolean generator. *)

float_range : float -> float -> float t (** eg: float_range 0.0 1.0 *)
```

```
val nat : int t        (** Generates small natural numbers. *)

val big_nat : int t    (** Generates natural numbers, possibly large. *)

val neg_int : int t    (** Generates 0 or negative ints. *)

val pint : int t       (** Generates 0 or positive integers uniformly. *)

val int : int t        (** Generates integers uniformly. *)
```

```
val oneof : 'a t list -> ‘a t

val frequency : (int * 'a t) list -> 'a t
```

```
val tup2 : 'a t -> 'b t -> ('a * 'b) t

val pair : 'a t -> 'b t -> ('a * 'b) t
```

The ‘a Gen.t module

```
val nat : int t          (** Generates small natural numbers. *)

val big_nat : int t      (** Generates natural numbers, possibly large. *)

val neg_int : int t      (** Generates 0 or negative ints. *)

val pint : int t         (** Generates 0 or positive integers uniformly. *)

val int : int t          (** Generates integers uniformly. *)
```

```
val oneof : 'a t list -> 'a t

val frequency : (int * 'a t) list -> 'a t
```

```
val tup2 : 'a t -> 'b t -> ('a * 'b) t

val pair : 'a t -> 'b t -> ('a * 'b) t
```

```
open Q.Gen

let pairs_of_pairs_gen = tup2 (tup2 nat nat) (tup2 int bool)

let big_and_small_gen = oneof [big_nat; nat]
```

The ‘a Gen.t module

```
open Q.Gen

let pairs_of_pairs = tup2 (tup2 nat nat) (tup2 int bool)

let big_and_small = oneof [big_nat; nat]
```

Interface so far gives:

- Some fixed ways to “get started” (base generators)
  - `nat, bool, int`
- Ways to put together base generators
  - `tup2, oneof`

What if we want our own way to get started?

What if we want to write our own generation algorithm?

A way to “get started”

A generator that does nothing but return a fixed value

```
val pure  : 'a -> 'a t
```

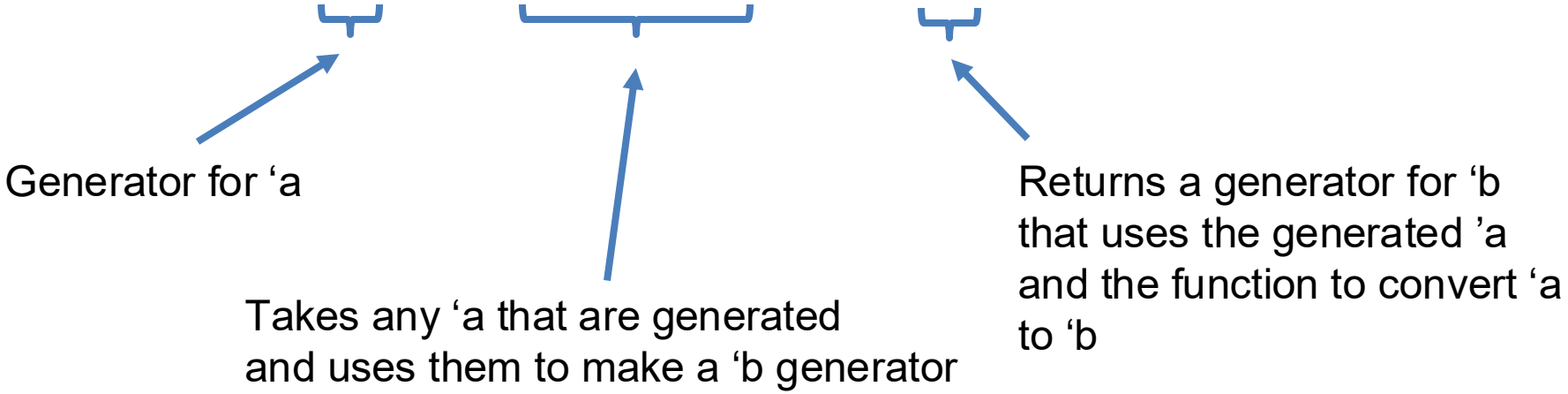
```
let three : int t = pure 3      (* always returns 3 *)
let tru : bool t = pure true    (* always returns true *)
```

```
type color = Green | Red | White

let green : color t = pure Green
let red   : color t = pure Red
let white : color t = pure White

let colors : color generator = oneof [green; red; white]
```

”Algorithms” are nothing but compositions of basic things

Generate then do a function

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```



”Algorithms” are nothing but compositions of basic things

```
let zero_one : float t =
  float_range 0.0 1.0

(* given a float, generated a bool *)
let coin : float -> bool t =
  fun f ->
     if x < .7 then
       pure true
     else
       pure false

(* a weighted coin generator *)
let coin = zero_one >>= coin
```

”Algorithms” are nothing but compositions of basic things

```
type tree = Leaf | Node of int * tree * tree

val insert : int -> tree -> tree

let gentree : tree arb =
  let open G in
  let rec aux (size:int) (t:tree) : tree t =
    if size <= 0 then pure t
    else
      int_bound 10 >>= fun v ->
      aux (size - 1) (insert v t)
  in
  Q.make ~print:tree_to_string
         (int_bound 10 >>= fun size -> aux size Leaf)
```

”Algorithms” are nothing but compositions of basic things

```
type tree = Leaf | Node of int * tree * tree

val insert : int -> tree -> tree

let gentree : tree arb =
  let open G in
  let rec aux (size:int) : tree t =
    if size <= 0 then pure Leaf
    else
      frequency
        [ (1, pure Leaf);
          (4,
             int >>= fun v ->
             aux (size / 2) >>= fun l ->
             aux (size / 2) >>= fun r ->
             pure (Node(v, l, r)))
```

”Algorithms” are nothing but compositions of basic things

```
val int_bound : int -> int t

let genL xs = int_bound (List.length xs)
let get xs n = pure (List.nth n elems)

let oneof (xs : 'a list) : 'a t =
  genL xs >= (fun n -> get xs n)
```

```
val map : ('a -> 'b) -> 'a t -> 'b t

let oneof (xs : 'a list) : 'a t =
  map (fun n -> List.nth n elems) (genL xs)
```

Map vs Bind

```
val pure  : 'a -> 'a t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

```
val map   : ('a -> 'b) -> 'a t -> 'b t
```

```
val pint : int t
val negint : int t

let pos_or_neg =
  bool >= (fun b -> if b then pint else negint)
```

Can't easily use map with pre-existing generators …

But you can code map if you already have >>= (try yourself)

Implementing Map and >>=

```
val pure  : 'a -> 'a t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

```
val map   : ('a -> 'b) -> 'a t -> 'b t
```

Recall:

```
type 'a t = Random.State -> 'a
```

Implementing Map and >>=

```
val pure  : 'a -> 'a t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

```
val map   : ('a -> 'b) -> 'a t -> 'b t
```

Recall:

```
type 'a t = Random.State -> 'a
```

```
let pure (x:'a) : 'a t = fun st -> x
```

Implementing Map and >>=

```
val pure  : 'a -> 'a t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

```
val map   : ('a -> 'b) -> 'a t -> 'b t
```

Recall:

```
type 'a t = Random.State -> 'a
```

```
let (>>=) (g:'a t) (f : 'a -> 'b t) : 'b t =
  fun st ->
    let a = g st in (f a) st
```

Implementing Map and >>=

```
val pure  : 'a -> 'a t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

```
val map   : ('a -> 'b) -> 'a t -> 'b t
```

Recall:

```
type 'a t = Random.State -> 'a
```

```
let (>>=) (g:'a t) (f : 'a -> 'b t) : 'b t =
  fun st ->
    let a = g st in (f a) st
```

```
let map (f : 'a -> 'b) (g:'a t) : 'b t =
  fun st ->
    let a = g st in (f a)
```

Monads

A data structure with an interface that includes

```
val pure  : 'a -> 'a t          (* AKA: return *)
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t   (* AKA: bind *)
```

where pure and >>= satisfy certain equational laws is called a **Monad**

in our case, 'a t = Random.State -> 'a but there are other definitions of 'a t used for other purposes

- 'a t = 'a option - the error monad
- 'a t = 'a list - the non-determinism monad
- 'a t = state -> 'a state - the "state" monad

In general, monads can be used to include state in a "safe" way inside otherwise totally functional languages.

This is how Haskell is designed. https://www.haskell.org/

Monads

A data structure with an interface that includes

```
val pure  : 'a -> 'a t
```

```
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
```

where pure and >>= satisfy certain equational laws is called a **Monad**

Monad Laws (for values of the right type)

```
pure v >>= f          = f v

m  >>= return         = m

(m >>= g) >>= h       = m >>= (fun x -> g x >>= h)
```

these laws can be used to help you reason about monadic programs,
just like knowing commutativity/associativity of + helps you reason about arithmetic

Summary

Be lazy! Don't write tests, generate them!

Use property-based thinking to gain deeper insight into the requirements

PBT Resources

Search for "property-based testing" and your language!

- Java: jqwik.net
- Python: https://hypothesis.readthedocs.io/en/latest/

Look for talks by John Hughes

- How to specify it! https://www.youtube.com/watch?v=zvRAyq5wj38
- Don't write tests! https://www.youtube.com/watch?v=hXnS_Xjwk2Y

And others may be good

- "Property-BasedTesting in a Screencast Editor" by OskarWickström
- "MetamorphicTesting" by HillelWayne