SOFTWARE FOUNDATIONS

VOLUME 4: QUICKCHICK: PROPERTY-BASED TESTING IN COQ

TABLE OF CONTENTS INDEX

QC

CORE QUICKCHICK

Generators

The heart of property-based random testing is generation of random inputs.

The G Monad

In QuickChick, a generator for elements of some type `A` belongs to the type `G A`. Intuitively, this type describes functions that take a random seed to an element of `A`. We will see below that `G` is actually a bit more than this, but this intuition will do for now.

QuickChick provides a number of primitives for building generators. First, `returnGen` takes a constant value and yields a generator that always returns this value.

```
Check returnGen.
```

the syntax P? takes a proposition (could be type) and turns it into a boolean expression

```
===> returnGen : ?A -> G ?A
```

We can see how it behaves by using the `Sample` command, which supplies its generator argument with several different random seeds:

```
(* Sample (returnGen 42). *)

===> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

Next, given a random generator for `A` and a *function* `f` taking an `A` and yielding a random generator for `B`, we can plumb the two together into a generator for `B` that works by internally generating a random `A` and applying `f` to it.

```
Check bindGen.

===> bindGen : G ?A -> (?A -> G ?B) -> G ?B
```

Naturally, the implementation of `bindGen` must take care to pass different random seeds to the two sub-generators!

With these two primitives in hand, we can make `G` an instance of the `Monad` typeclass.

```
Section GMonadDef.

Instance GMonad : `{Monad G} | 3 :=
  {
    ret := @returnGen;
    bind := @bindGen
  }.

End GMonadDef.
```

Primitive generators

QuickChick provides several primitive generators for "ordered types," accessed via the `choose` combinator.

```
Check @choose.

  ===>
  @choose
  : forall A : Type, ChoosableFromInterval A -> A * A -> G A
```

The `ChoosableFromInterval` typeclass describes primitive types `A`, like natural numbers and integers (`Z`), for which it makes sense to randomly generate elements from a given interval.

```
Print ChoosableFromInterval.

  ===>
  Record ChoosableFromInterval (A : Type) : Type :=
    Build_ChoosableFromInterval
    { super : Ord A;
      randomR : A * A -> RandomSeed -> A * RandomSeed;
      ...
    }.
(* Sample (choose (0,10)). *)

  ===> [ 1, 2, 1, 9, 8, 1, 3, 6, 2, 1, 8, 0, 1, 1, 3, 5, 4, 10, 4, 6 ]
```

Exercise: 1 star, optional (cfi)

discuss!!

Print out the full definition of `ChoosableFromInterval`. Can you understand what it means? □

Lists

Since they are a very commonly used compound datatype, lists have special combinators in QuickChick: `listOf` and `vectorOf`.

The `listOf` combinator takes as input a generator for elements of type `A` and returns a generator for lists of `A`s.

```
Check listOf.

  ===> listOf : G ?A -> G (list ?A)
(* Sample (listOf (choose (0,4))). *)
```

```
===>
 [ [ 0, 3, 2, 0 ],
   [ 1, 3, 4, 1, 0, 3, 0, 2, 2, 3, 2, 2, 2, 0, 4, 2, 3, 0, 1 ],
   [ 3, 4, 3, 1, 2, 4, 4, 1, 0, 3, 4, 3, 2, 2, 4, 4, 1 ],
   [ 0 ],
   [ 4, 2, 3 ],
   [ 3, 3, 4, 0, 1, 4, 3, 2, 4, 1 ],
   [ 0, 4 ],
   [  ],
   [ 1, 0, 1, 3, 1 ],
   [ 0, 0 ],
   [ 1, 4 ],
   [ 4, 3, 2, 0, 2, 2, 4, 0 ],
   [ 1, 1, 0, 0, 1, 4 ],
   [ 2, 0, 2, 1, 3, 3 ],
   [ 4, 3, 3, 0, 1 ],
   [ 3, 3, 3 ],
   [ 3, 2, 4 ],
   [ 1, 2 ],
   [  ],
   [  ] ]
```

The second combinator, `vectorOf`, receives an additional numeric argument `n`, the length of the list to be generated.

```
Check vectorOf.

  ===> vectorOf : nat -> G ?A -> G (list ?A)
(* Sample (vectorOf 3 (choose (0,4))). *)

  ===>
 [ [0, 1, 4],
   [1, 1, 0],
   [3, 3, 3],
   [0, 2, 1],
   [1, 3, 2],
   [3, 3, 0],
   [3, 0, 4],
   [2, 3, 3],
   [3, 2, 4],
   [1, 2, 3],
   [2, 0, 4]  ]
```

This raises a question. It's clear how `vectorOf` decides how big to make its lists (we tell it!), but how does `listOf` do it? The answer is hidden inside `G`.

In addition to handling random-seed plumbing, the `G` monad also maintains a "current maximum size" (in the style of a "reader monad", if you like that terminology): a natural

number that can be used as an upper bound on the depth of generated objects.

Internally, `G A` is just a synonym for `nat → RandomSeed → A`.

```
Module DefineG.

Inductive G (A:Type) : Type :=
| MkG : (nat → RandomSeed → A) → G A.

End DefineG.
```

When it is searching for counterexamples, QuickChick progressively tries larger and larger values for the size bound `n`, to gradually explore deeper into the search space.

Each generator can choose to interpret the size bound however it wants, and there is no *enforced* guarantee that generators pay any attention to it at all; however, it is good practice to respect this bound when programming generators.

Custom Generators

Naturally, we also need generators for user-defined datatypes. Here's a simple one to play with:

```
Inductive color := Red | Green | Blue | Yellow.
```

In order for commands like `Sample` to display colors, we should make `color` an instance of the `Show` typeclass:

```
Require Import String. Open Scope string.
Instance show_color : Show color :=
  {| show c :=
       match c with
         | Red ⇒ "Red"
         | Green ⇒ "Green"
         | Blue ⇒ "Blue"
         | Yellow ⇒ "Yellow"
       end
  |}.
```

To generate a random color, we just need to pick one of the constructors `Red`, `Green`, `Blue`, or `Yellow`. This is done using `elems_`.

```
Check elems_.

  ===> elems_ : ?A -> list ?A -> G ?A

Definition genColor' : G color :=
  elems_ Red [ Red ; Green ; Blue ; Yellow ].

(* Sample genColor'. *)

  ===>
 [Red, Green, Blue, Blue, Red, Yellow, Blue, Red, Blue, Blue, Red]
```

why does it need to take an element of type A? answer:

The first argument to `elems_` serves as a default result. If its list argument is not empty, `elems_` returns a generator that always picks an element of that list; otherwise the generator always returns the default object. This makes Coq's totality checker happy, but

makes `elem_` a little awkward to use, since typically its second argument will be a non-empty constant list.

To make the common case smoother, QuickChick provides convenient notations that automatically extract the default element.

```
" 'elems' [ x ] "
    := elems_ x (cons x nil)
" 'elems' [ x ; y ] "
    := elems_ x (cons x (cons y nil))
" 'elems' [ x ; y ; .. ; z ] "
    := elems_ x (cons x (cons y (.. (cons z nil))))
" 'elems' ( x ;; l ) "
    := elems_ x (cons x l)
```

Armed with `elems`, we can write a `color` generator the way we'd hope.

```
Definition genColor : G color :=
  elems [ Red ; Green ; Blue ; Yellow ].

(* Sample genColor. *)
```

```
===> [Red, Green, Blue, Blue, Red, Yellow, Blue, Red, Blue, Blue, Red]
```

For more complicated ADTs, QuickChick provides more combinators.

We showcase these using everyone's favorite datatype: trees!

Our trees are standard polymorphic binary trees; either `Leaf`s or `Node`s containing some payload of type `A` and two subtrees.

```
Inductive Tree A :=
| Leaf : Tree A
| Node : A → Tree A → Tree A → Tree A.

Arguments Leaf {A}.
Arguments Node {A} _ _ _.
```

Before getting to generators for trees, we again give a straightforward `Show` instance. (The need for a local `let fix` declaration stems from the fact that Coq's typeclasses, unlike Haskell's, are not automatically recursive. We could alternatively define `aux` with a separate top-level `Fixpoint`.)

```
Instance showTree {A} `{_ : Show A} : Show (Tree A) :=
  {| show := let fix aux t :=
       match t with
         | Leaf ⇒ "Leaf"
         | Node x l r ⇒
             "Node (" ++ show x ++ ") ("
                      ++ aux l ++ ") ("
                      ++ aux r ++ ")"
       end
     in aux
  |}.
```

This brings us to our first generator *combinator*, called `oneOf_`.

```
Check oneOf_.
```

```
===> oneOf_ : G ?A -> list (G ?A) -> G ?A
```

This combinator takes a default generator and a list of generators, and it picks one of the generators from the list uniformly at random (unless the list is empty, in which case it picks from the default generator). As with `elems_`, QuickChick introduces a more convenient notation `oneOf` to hide this default element.

Next, Coq's termination checker will save us from shooting ourselves in the foot!

The "obvious" first attempt at a generator is the following function `genTree`, which generates either a `Leaf` or else a `Node` whose subtrees are generated recursively (and whose payload is produced by a generator `g` for elements of type `A`).

```
Fixpoint genTree {A} (g : G A) : G (Tree A) :=
  oneOf [ ret Leaf ;;
          liftM3 Node g (genTree g) (genTree g) ].
```

Of course, this fixpoint will not pass Coq's termination checker. Attempting to justify this fixpoint informally, one might first say that at some point the random generation will pick a `Leaf` so it will eventually terminate. But the termination checker cannot understand this kind of probabilistic reasoning. Moreover, even informally, the reasoning is wrong: Every time we choose to generate a `Node`, we create two separate branches that must both be terminated with leaves. It is not hard to show that the *expected* size of the generated trees is actually infinite!

the FUEL idiom

The solution is to use the standard "fuel" idiom that all Coq users know. We add a natural number `sz` as a parameter. We decrease this size in each recursive call, and when it reaches `O`, we always generate `Leaf`. Thus, the initial `sz` parameter serves as a bound on the depth of the tree.

```
Fixpoint genTreeSized {A} (sz : nat) (g : G A) : G (Tree A) :=
  match sz with
    | O ⇒ ret Leaf
    | S sz' ⇒
      oneOf [
        ret Leaf ;
        liftM3 Node g (genTreeSized sz' g) (genTreeSized sz' g)
      ]
  end.

(* Sample (genTreeSized 3 (choose(0,3))). *)

  ===>
    [ Leaf,
      Leaf,
      Node (3) (Node (0) (Leaf) (Leaf))
               (Node (2) (Leaf) (Node (3) (Leaf) (Leaf))),
      Leaf,
      Leaf,
      Leaf,
      Node (1) (Leaf) (Node (1) (Leaf) (Node (0) (Leaf) (Leaf))),
      Leaf,
```

```
      Node (3) (Leaf) (Leaf),
      Node (1) (Leaf) (Leaf),
      Leaf,
      Leaf,
      Node (0) (Leaf) (Node (0) (Leaf) (Node (2) (Leaf) (Leaf))),
      Node (0) (Node (2) (Node (3) (Leaf) (Leaf)) (Leaf)) (Leaf),
      Node (0) (Leaf) (Leaf),
      Leaf,
      Leaf,
      Leaf,
      Leaf,
      Leaf ]
```

While this generator succeeds in avoiding nontermination, we can see just by observing the result of `Sample` that there is a problem: it produces way too many `Leaf`s! This is actually not surprising, since half the time we generate a `Leaf` right at the outset.

We can obtain bigger trees more often if we skew the distribution towards `Node`s using a more expressive QuickChick combinator, `freq_`.

```
Check freq_.

===> freq_ : G ?A -> seq (nat * G ?A) -> G ?A
```

As with `oneOf`, we usually use a convenient derived notation, called `freq`, that takes a list of generators, each tagged with a natural number that serves as the weight of that generator. For example, in the following generator, a `Leaf` will be generated 1 / (sz + 1) of the time and a `Node` the remaining sz / (sz + 1) of the time.

```
Fixpoint genTreeSized' {A} (sz : nat) (g : G A) : G (Tree A) :=
  match sz with
    | O ⇒ ret Leaf
    | S sz' ⇒
      freq [ (1, ret Leaf) ;
             (sz, liftM3 Node g (genTreeSized' sz' g)
                                (genTreeSized' sz' g))
           ]
  end.

(* Sample (genTreeSized' 3 (choose(0,3))). *)

===>
    [ Node (3) (Node (1) (Node (3) (Leaf) (Leaf)) (Leaf))
               (Node (0) (Leaf) (Node (3) (Leaf) (Leaf))),
      Leaf,
      Node (2) (Node (1) (Leaf) (Leaf)) (Leaf),
      Node (0) (Leaf) (Node (0) (Node (2) (Leaf) (Leaf))
                                (Node (0) (Leaf) (Leaf))),
      Node (1) (Node (2) (Leaf) (Node (0) (Leaf) (Leaf))) (Leaf),
      Node (0) (Node (0) (Leaf) (Node (3) (Leaf) (Leaf)))
               (Node (2) (Leaf) (Leaf)),
      Node (1) (Node (3) (Node (2) (Leaf) (Leaf)) (Node (3) (Leaf) (Leaf)))
```

```
                (Node (1) (Leaf) (Node (2) (Leaf) (Leaf))),
          Node (0) (Node (0) (Node (0) (Leaf) (Leaf)) (Node (1) (Leaf) (Leaf)))
                (Node (2) (Node (3) (Leaf) (Leaf)) (Node (0) (Leaf) (Leaf))),
          Node (2) (Node (2) (Leaf) (Leaf)) (Node (1) (Node (2) (Leaf) (Leaf))
                                                      (Node (2) (Leaf) (Leaf))),
          Node (2) (Node (3) (Node (2) (Leaf) (Leaf)) (Leaf))
                (Node (0) (Node (2) (Leaf) (Leaf)) (Leaf)),
          Leaf,
          Node (2) (Node (3) (Node (3) (Leaf) (Leaf)) (Leaf)) (Leaf),
          Leaf,
          Node (1) (Leaf) (Leaf),
          Leaf,
          Node (1) (Node (2) (Leaf) (Node (3) (Leaf) (Leaf)))
                (Node (0) (Leaf) (Node (1) (Leaf) (Leaf))),
          Leaf,
          Node (3) (Node (0) (Node (0) (Leaf) (Leaf)) (Leaf))
                (Node (0) (Leaf) (Node (2) (Leaf) (Leaf))),
          Node (2) (Node (2) (Node (0) (Leaf) (Leaf)) (Leaf))
                (Node (1) (Leaf) (Node (2) (Leaf) (Leaf))),
          Leaf ]
```

This looks better.

Exercise: 2 stars (genListSized)

Write a sized generator for lists, following `genTreeSized'`.

```
  (* FILL IN HERE *)
```
□

Exercise: 3 stars (genColorOption)

Write a custom generator for values of type `option color`. Make it generate `None` about 1/10th of the time, and make it generate `Some Red` three times as often as the other colors.

```
  (* FILL IN HERE *)
```
□

Checkers

To showcase how such generators can be used to find counterexamples, suppose we define a function for "mirroring" a tree — swapping its left and right subtrees recursively.

```
Fixpoint mirror {A : Type} (t : Tree A) : Tree A :=
  match t with
    | Leaf ⇒ Leaf
    | Node x l r ⇒ Node x (mirror r) (mirror l)
  end.
```

To formulate a property about `mirror`, we need a structural equality test on trees. We can obtain that with minimal effort using the `Dec` typeclass and the `dec_eq` tactic.

```
Instance eq_dec_tree (t1 t2 : Tree nat) : Dec (t1 = t2) := {}.
Proof. dec_eq. Defined.
```

We expect that mirroring a tree twice should yield the original tree.

```
Definition mirrorP (t : Tree nat) := (mirror (mirror t)) = t?.
```

Now we want to use our generator to create a lot of random trees and, for each one, check whether `mirrorP` returns `true` or `false`.

Another way to say this is that we want to use `mirrorP` to build a *generator for test results*.

Let's see how this works.

(Let's open a playground module so we can show simplified versions of the actual QuickChick definitions.)

```
Module CheckerPlayground1.
```

First, we need a type of test results — let's call it `Result`. For the moment, think of `Result` as just an enumerated type with constructors `Success` and `Failure`.

```
Inductive Result := Success | Failure.

Instance showResult : Show Result :=
  {
    show r := match r with Success ⇒ "Success" | Failure ⇒ "Failure"
end
  }.
```

Then we can define the type `Checker` to be `G Result`.

```
Definition Checker := G Result.
```

That is, a `Checker` embodies some way of performing a randomized test of the truth of some proposition, which, when applied to a random seed, will yield either `Success` (meaning that the proposition survived this test) or `Failure` (meaning that this test demonstrates that the proposition was false).

Sampling a `Checker` many times with different seeds will cause many different tests to be performed.

To check `mirrorP`, we'll need a way to build a `Checker` out of a function from trees to booleans.

Actually, we will be wanting to build `Checker`s based on many different types. So let's begin by defining a typeclass `Checkable`, where an instance for `Checkable A` will provide a way of converting an `A` into a `Checker`.

```
Class Checkable A :=
  {
    checker : A → Checker
  }.
```

It is easy to give a `bool` instance for `Checkable`.

```
Instance checkableBool : Checkable bool :=
  {
    checker b := if b then ret Success else ret Failure
  }.
```

That is, the boolean value `true` passes every test we might subject it to (i.e., the result of `checker` is a `G` that returns `true` for every seed), while `false` fails all tests.

Let's see what happens if we sample checkers for our favorite booleans, `true` and `false`.

(We need to exit the playground so that we can do `Sample`: because `Sample` is implemented internally via extraction to OCaml, which usually does not work from inside a `Module`.)

```
End CheckerPlayground1.

(* Sample (CheckerPlayground1.checker true). *)

  ===>
   [Success, Success, Success, Success, Success, Success, Success,
    Success, Success, Success, Success]
(* Sample (CheckerPlayground1.checker false). *)

  ===>
   [Failure, Failure, Failure, Failure, Failure, Failure, Failure,
    Failure, Failure, Failure, Failure]
```

What we've done so far may look a a bit strange, since these checkers always generate the same results. We'll make things more interesting in a bit, but first let's pause to define one more basic instance of `Checkable`.

A decidable `Prop` is not too different from a boolean, so we should be able to build a checker from that too.

```
Module CheckerPlayground2.
Export CheckerPlayground1.

Instance checkableDec `{P : Prop} `{Dec P} : Checkable P :=
  {
    checker p := if P? then ret Success else ret Failure
  }.
```

(The definition looks a bit strange since it doesn't use its argument `p`. The intuition is that all the information in `p` is already encoded in `P`!)

Now suppose we pose a couple of (decidable) conjectures:

an opinion or conclusion formed on the basis of incomplete information.

```
Conjecture c1 : 0 = 42.
Conjecture c2 : 41 + 1 = 42.

End CheckerPlayground2.
```

The somewhat astononishing thing about the `Checkable` instance for decidable `Prop`s is that, even though these are *conjectures* (we haven't proved them, so the "evidence" that Coq has for them internally is just an uninstantiated "evar"), we can still build checkers from them and sample from these checkers! (Why? Technically, it is because the `Checkable` instance for decidable properties does not look at its argument.)

```
(*  Sample (CheckerPlayground1.checker CheckerPlayground2.c1). *)
```

```
===>
 [Failure, Failure, Failure, Failure, Failure, Failure, Failure,
  Failure, Failure, Failure, Failure]
(* Sample (CheckerPlayground1.checker CheckerPlayground2.c2). *)

===>
 [Success, Success, Success, Success, Success, Success, Success,
  Success, Success, Success, Success]
```

Again, the intuition is that, although we didn't present proofs (and could not have, in the first case!), Coq already "knows" either a proof or a disproof of each of these conjectures because they are decidable.

```
Module CheckerPlayground3.
Import CheckerPlayground2.
```

Now let's go back to `mirrorP`.

We have seen that the result of `mirrorP` is `Checkable`. What we need is a way to take a function returning a checkable thing and make the function itself checkable.

We can easily do this, as long as the argument type of the function is something we know how to generate!

```
Definition forAll {A B : Type} `{Checkable B}
                  (g : G A) (f : A → B)
                : Checker :=
    a <- g ;;
    checker (f a).
```

kind of confused!!!

```
End CheckerPlayground3.
```

Let's try this out. We can define a boolean test that returns `true` for `Red` and `false` for other colors.

```
Definition isRed c :=
  match c with
    Red ⇒ true
  | _ ⇒ false
  end.
```

Since we can generate elements of `color` and we have a `Checkable` instance for `bool`, we can apply `forAll` to `isRed` and sample from the resulting `Checker` to run some tests.

```
(* Sample (CheckerPlayground3.forAll genColor isRed). *)

===>
 [Success, Failure, Failure, Failure, Success, Failure,
  Failure, Success, Failure, Failure, Success]
```

Looks like not all colors are `Red`.

Good to know.

Now, what about `mirrorP`?

```
(*
Sample (CheckerPlayground3.forAll
          (genTreeSized' 3 (choose(0,3)))
          mirrorP).
*)

   ===>
    [Success, Success, Success, Success, Success, Success, Success,
     Success, Success, Success, Success]
```

Excellent: It looks like many tests are succeeding — maybe the property is true.

Let's instead try defining a bad property and see if we can detect that it's bad...

```
Definition faultyMirrorP (t : Tree nat) := (mirror t) = t ?.

(*
Sample (CheckerPlayground3.forAll
          (genTreeSized' 3 (choose(0,3)))
          faultyMirrorP).
*)

   ===>
    [Failure, Success, Failure, Success, Success, Success, Failure,
     Success, Failure, Failure, Success]
```

Great — looks like a good number of tests are failing now, as expected.

There's only one little issue: What *are* the tests that are failing? We can tell by looking at the samples that the property is bad, but we can't see the counterexamples!

We can fix this by going back to the beginning and enriching the `Result` type to keep track of failing counterexamples.

```
Module CheckerPlayground4.

Inductive Result :=
  | Success : Result
  | Failure : ∀ {A} `{Show A}, A → Result.

Instance showResult : Show Result :=
  {
    show r := match r with
                Success ⇒ "Success"
              | Failure A showA a ⇒ "Failure: " ++ show a
              end
  }.

Definition Checker := G Result.

Class Checkable A :=
  {
    checker : A → Checker
  }.

Instance showUnit : Show unit :=
  {
    show u := "tt"
  }.
```

The failure cases in the `bool` and `Dec` checkers don't need to record anything except the `Failure`, so we put `tt` (the sole value of type `unit`) as the "failure reason."

```
Instance checkableBool : Checkable bool :=
  {
    checker b := if b then ret Success else ret (Failure tt)
  }.

Instance checkableDec `{P : Prop} `{Dec P} : Checkable P :=
  {
    checker p := if P? then ret Success else ret (Failure tt)
  }.
```

The interesting case is the `forAll` combinator. Here, we *do* have some useful information to record in the failure case — namely, the argument that caused the failure.

```
Definition forAll {A B : Type} `{Show A} `{Checkable B}
                  (g : G A) (f : A → B)
                : Checker :=
  a <- g ;;
  r <- checker (f a) ;;
  match r with
    Success ⇒ ret Success
  | Failure B showB b ⇒ ret (Failure (a,b))
  end.
```

Note that, rather than just returning `Failure a`, we package up `a` together with `b`, which explains the reason for the failure of `f a`. This allows us to write several `forAll`s in sequence and capture all of their results in a nested tuple.

```
End CheckerPlayground4.

(*
Sample (CheckerPlayground4.forAll
          (genTreeSized' 3 (choose(0,3)))
          faultyMirrorP).
*)
```

```
   ===>
    [Failure: (Node (2) (Node (3) (Node (2) (Leaf) (Leaf)) (Leaf))
                        (Node (0) (Node (2) (Leaf) (Leaf)) (Leaf)), tt),
     Success,
     Failure: (Node (2) (Node (3) (Node (3) (Leaf) (Leaf)) (Leaf)) (Leaf), tt),
     Success, Success, Success,
     Failure: (Node (1) (Node (2) (Leaf) (Node (3) (Leaf) (Leaf)))
                        (Node (0) (Leaf) (Node (1) (Leaf) (Leaf))), tt),
     Success,
     Failure: (Node (3) (Node (0) (Node (0) (Leaf) (Leaf)) (Leaf))
                        (Node (0) (Leaf) (Node (2) (Leaf) (Leaf))), tt),
     Failure: (Node (2) (Node (2) (Node (0) (Leaf) (Leaf)) (Leaf))
                        (Node (1) (Leaf) (Node (2) (Leaf) (Leaf))), tt),
     Success]
```

The bug is found several times and actual counterexamples are reported: nice!

Sampling repeatedly from a generator is just what the `QuickChick` command does, except that, instead of running a fixed number of tests and returning their results in a list, it runs tests only until the first counterexample is found.

```
(*
QuickChick
  (forAll
     (genTreeSized' 3 (choose(0,3)))
     faultyMirrorP).
*)

   ===>
   QuickChecking (forAll (genTreeSized' 3 (choose (0, 3))) faultyMirrorP)

   Node (0) (Node (0) (Node (2) (Leaf) (Leaf))
                      (Node (1) (Leaf) (Leaf)))
            (Node (1) (Node (0) (Leaf) (Leaf)) (Leaf))

   *** Failed after 1 tests and 0 shrinks. (0 discards)
```

However, these counterexamples themselves still leave something to be desired: they are all *much* larger than is really needed to illustrate the bad behavior of `faultyMirrorP`.

This is where *shrinking* comes in.

Shrinking

Shrinking (sometimes known as "delta debugging") is a process that, given a counterexample to some property, searches (greedily) for smaller counterexamples.

Given a shrinking function `s` of type `A → list A` and a value `x` of type `A` that is known to falsify some property `P`, QuickChick tries `P` on all members of `s x` until it finds another counterexample. It repeats this process, starting from the new counterexample, until it reaches a point where `x` fails property `P` but every element of `s x` succeeds. This `x` is a "locally minimal" counterexample.

be mindful of the total order!!

Clearly, this greedy algorithm only work if all elements of `s x` are strictly "smaller" than `x` for all `x` — that is, there should be some total order on the type of `x` such that `s` is strictly decreasing in this order. This is the basic correctness requirement for a shrinker. It is guaranteed by all the shrinkers that QuickChick derives automatically, but ifyou roll your own shrinkers you need to be careful to maintain it.

Here is a shrinker for `colors`.

```
Instance shrinkColor : Shrink color :=
  {
    shrink c :=
      match c with
      | Red ⇒ [ ]
      | Green ⇒ [ Red ]
      | Blue ⇒ [ Red; Green ]
      | Yellow ⇒ [ Red; Green; Blue ]
```

```
      end
  }.
```

Most of the time, shrinking functions should try to return elements that are "just one step" smaller than the one they are given. For example, consider the default shrinking function for lists provided by QuickChick.

```
Print shrinkList.

  ===>
    shrinkList =
      fun (A : Type) (H : Shrink A) =>
         {| shrink := shrinkListAux shrink |}
    : forall A : Type, Shrink A -> Shrink (list A)


Print shrinkListAux.

  ===>
    shrinkListAux =
      fix shrinkListAux (A : Type) (shr : A -> list A) (l : list A) :
        list (list A) :=
        match l with
        | [] => []
        | x :: xs =>
            ((xs :: map (fun xs' : list A => x :: xs')
                        (shrinkListAux A shr xs)) ++
             map (fun x' : A => x' :: xs) (shr x))
        end
    : forall A : Type, (A -> list A) -> list A -> list (list A)
```

An empty list cannot be shrunk, as there is no smaller list. A cons cell can be shrunk in three ways: by returning the tail of the list, by shrinking the tail of the list and keeping the same head, or by shrinking the head and keeping the same tail. By induction, this process can generate all smaller lists.

(Pro tip: One refinement to the advice that shrinkers should return "one step smaller" values is that, for potentially large data structures, it can sometimes greatly improve performance if a shrinker returns not only all the "slightly smaller" structures but also one or two "much smaller" structures. E.g. for a number `x` greater than `2`, we might shrink to `0; x div 2; pred x`.)

Writing a shrinking instance for trees is equally straightforward: we don't shrink `Leaf`s, while for `Node`s we can either return the left or right subtree, or shrink the payload, or shrink one of the subtrees.

```
Open Scope list.
Fixpoint shrinkTreeAux {A}
             (s : A → list A) (t : Tree A)
           : list (Tree A) :=
  match t with
    | Leaf ⇒ []
    | Node x l r ⇒ [l] ++ [r] ++
```

```
                      map (fun x' ⇒ Node x' l r) (s x) ++
                      map (fun l' ⇒ Node x l' r) (shrinkTreeAux s l)
++
                      map (fun r' ⇒ Node x l r') (shrinkTreeAux s r)
  end.

Instance shrinkTree {A} `{Shrink A} : Shrink (Tree A) :=
  {| shrink x := shrinkTreeAux shrink x |}.
```

With `shrinkTree` in hand, we can use the `forAllShrink` property combinator, a variant of `forAll` that takes a shrinker as an additional argument, to test properties like `faultyMirrorP`.

```
(*
  QuickChick
       (forAllShrink
          (genTreeSized' 5 (choose (0,5)))
          shrink
          faultyMirrorP).
*)

   ===>
     Node (0) (Leaf) (Node (0) (Leaf) (Leaf))


     *** Failed! After 1 tests and 8 shrinks
```

We now get a quite simple counterexample (in fact, one of two truly minimal ones), from which it is easy to see that the bad behavior occurs when the subtrees of a `Node` are different.

Exercise: Ternary Trees

For a comprehensive but still fairly simple exercise involving most of what we have seen, let's consider ternary trees, whose nodes have three childen instead of two.

```
Inductive TernaryTree A :=
| TLeaf : TernaryTree A
| TNode :
    A → TernaryTree A → TernaryTree A → TernaryTree A → TernaryTree
A.

Arguments TLeaf {A}.
Arguments TNode {A} _ _ _ _.
```

Also consider the following (faulty?) mirror function. We'll want to do some testing to see what we can find about this function.

```
Fixpoint tern_mirror {A : Type} (t : TernaryTree A) : TernaryTree A
:=
  match t with
    | TLeaf ⇒ TLeaf
    | TNode x l m r ⇒ TNode x (tern_mirror r) m (tern_mirror l)
  end.
```

Exercise: 1 star (show_tern_tree)

Write a `Show` instance for Ternary Trees.

```
  (* FILL IN HERE *)
```
□

Exercise: 2 stars (gen_tern_tree)

Write a generator for ternary trees.

```
  (* FILL IN HERE *)
```

The following line should generate a bunch of nat ternary trees.

```
  (* Sample (@genTernTreeSized nat 3 (choose (0,10))). *)
```
□

Exercise: 2 stars (shrink_tern_tree)

Write a shrinker for ternary trees.

```
  (* FILL IN HERE *)
```
□

From the root node, we can define a path in a ternary tree by first defining a direction corresponding to a choice of child tree...

```
Inductive direction := Left | Middle | Right.
```

A direction tells us which child node we wish to visit. The `traverse_node` function uses a `direction` to visit the corresponding child.

```
Definition traverse_node {A} (t:direction) (l m r: TernaryTree A) :=
  match t with
  | Left ⇒ l
  | Middle ⇒ m
  | Right ⇒ r
  end.
```

A `path` in a ternary tree is a list of `directions`.

```
Definition path := list direction.
```

We can traverse a path by iterating over the `directions` in the path and traversing the corresponding nodes.

```
Fixpoint traverse_path {A} (p:path) (t: TernaryTree A) :=
  match t with
  | TLeaf ⇒
    TLeaf
  | TNode x l m r ⇒
    match p with
    | h :: tl ⇒ traverse_path tl (traverse_node h l m r)
    | [] ⇒ t
    end
  end.
```

And we can mirror a path by simply swapping left and right throughout.

```
Definition path_mirror := map (fun t ⇒ match t with
                                       | Left ⇒ Right
                                       | Right ⇒ Left
                                       | _ ⇒ t
                                       end).
```

Before we can state some properties about paths in ternary trees, it's useful to have a decidable equality for ternary trees of `nat`s.

```
Instance eq_dec_tern_tree (t1 t2 : TernaryTree nat) : Dec (t1 = t2)
:= {}.
Proof. dec_eq. Defined.
```

Traversing a path in a tree should be the same as traversing the mirror of the path in the mirror of the tree, just with a mirrored tree result.

```
Definition tern_mirror_path_flip (t:TernaryTree nat) (p:path) :=
  ( traverse_path p t
   = tern_mirror (traverse_path (path_mirror p) (tern_mirror t))) ?.
```

Before using this property to test `tern_mirror`, we'll need to be able to generate, show, and shrink paths. To start, we'll derive those For `directions`. The `Show` and `G` definitions for `direction` are simple to write; the `shrink` will by default go left.

```
Instance showDirection : Show direction :=
  {| show x :=
       match x with
       | Left ⇒ "Left"
       | Middle ⇒ "Middle"
       | Right ⇒ "Right"
       end |}.

Definition genDirection : G direction :=
  elems [Left; Middle; Right].

Instance shrinkDirection : Shrink direction :=
  {| shrink d :=
       match d with
       | Left ⇒ []
       | Right | Middle ⇒ [Left]
       end |}.
```

For generating a list of paths, we'll use the built-in function `listOf`, which takes a generator for a type and generates lists of elements of that type. We do the same for `show` and `shrink` with `showList` and `shrinkList` respectively.

```
Instance showPath : Show path := showList.

Definition genPath : G path :=
  listOf genDirection.

Instance shrinkPath : Shrink path := shrinkList.
```

Exercise: 4 stars (bug_finding_tern_tree)

Using `genTernTreeSized` and `shrinkTernTree` find (and fix!) any bugs in `tern_mirror`.

```
(* FILL IN HERE *)
```

□

Putting it all Together

Now we've got pretty much all the basic machinery we need, but the way we write properties — using `forAllShrink` and explicitly providing generators and shrinkers — is still heavier than it needs to be. We can use a bit more typeclass magic to further lighten things.

First, we introduce a typeclass `Gen A` with a single operator `arbitrary` of type `G A`.

```
Module DefineGen.

Class Gen (A : Type) :=
  {
    arbitrary : G A
  }.

End DefineGen.
```

Intuitively, `Gen` is a way of uniformly packaging up the generators for various types so that we do not need to remember their individual names — we can just call them all `arbitrary`.

```
Instance gen_color : Gen color :=
  {
    arbitrary := genColor
  }.
```

Next, for convenience we package `Gen` and `Shrink` together into an `Arbitrary` typeclass that is a subclass of both.

```
Module DefineArbitrary.
Import DefineGen.

Class Arbitrary (A : Type) `{Gen A} `{Shrink A}.
```

(The empty "body" of `Arbitrary` is elided.)

```
End DefineArbitrary.
```

We can use the top-level `QuickChick` command on quantified propositions with generatable and decidable conclusions, stating just the property and letting the typeclass machinery figure out the rest.

For example, suppose we want to test this:

```
Conjecture every_color_is_red : ∀ c, c = Red.
```

Since we have already defined `Gen` and `Shrink` instances for `color`, we automatically get an `Arbitrary` instance. The `Gen` part is used by the checker instance for "forall" propositions to generate random `color` arguments.

To show that the conclusion is decidable, we need to define a `Dec` instance for equality on colors.

```
Instance eq_dec_color (x y : color) : Dec (x = y).
Proof. dec_eq. Defined.
```

Putting it all together:

```
(*  QuickChick every_color_is_red. *)

  ===>
    QuickChecking every_color_is_red
    Green
    *** Failed after 1 tests and 1 shrinks. (0 discards)
```

Sized Generators

Suppose we want to build an `Arbitrary` instance for trees.

We would like to use `genTreeSized'`; however, that generator takes an additional `size` argument.

Fortunately, since the `G` monad itself includes a size argument, we can "plumb" this argument into generators like `genTreeSized'`.

In other words, we can define an operator `sized` that takes a] sized generator and produces an unsized one.

```
Module DefineSized.
Import DefineG.

Definition sized {A : Type} (f : nat → G A) : G A :=
  MkG _
      (fun n r ⇒
         match f n with
           MkG g ⇒ g n r
         end).

End DefineSized.
```

To streamline assembling generators, it is convenient to introduce one more typeclass, `GenSized`, whose instances are sized generators.

```
Module DefineGenSized.

Class GenSized (A : Type) :=
  {
    arbitrarySized : nat → G A
  }.
```

We can then define a generic `Gen` instance for types that have a `GenSized` instance, using `sized`:

```
Instance GenOfGenSized {A} `{GenSized A} : Gen A :=
  {
    arbitrary := sized arbitrarySized
  }.

End DefineGenSized.
```

Now we can make a `Gen` instance for trees by providing just an implementation of `arbitrarySized`.

```
Instance genTree {A} `{Gen A} : GenSized (Tree A) :=
  {| arbitrarySized n := genTreeSized' n arbitrary |}.
```

Finally, with the `Arbitrary` instance for trees, we can supply just `faultyMirrorP` to the `QuickChick` command.

```
(* QuickChick faultyMirrorP. *)
```

Exercise: 2 stars (tern_tree_typeclasses)

Add typeclass instances for `GenSized` and `Shrink` so that you can `QuickChick tern_mirror_reverse` directly.

```
(* FILL IN HERE *)

(* QuickChick tern_mirror_reverse. *)
```

□

Automation

Writing `Show` and `Arbitrary` instances is usually not hard, but it can get tedious when we are testing code that involves many new `Inductive` type declarations. To streamline this process, `QuickChick` provides some automation for deriving such instances for "plain datatypes" automatically!

```
Derive Arbitrary for Tree.

  ===> GenSizedree is defined
  ===> ShrinkTree is defined

Print GenSizedTree.
Print ShrinkTree.

Derive Show for Tree.

   ===> ShowTree is defined

Print ShowTree.
```

Collecting Statistics

Earlier in this tutorial we observed that our first definition of `genTreeSized` seemed to be producing too many `Leaf` constructors.

In that case, just eyeballing at a few results from `Sample` gave us an idea that something was wrong with the distribution of test cases, but it's often useful to collect more extensive statistics from larger sets of samples.

This is where `collect`, another property combinator, comes in.

```
Check @collect.
```

```
===>
@collect
  : forall A prop : Type, Show A -> Checkable prop ->
      A -> prop -> Checker
```

That is, `collect` takes a checkable proposition and returns a new `Checker` (for the same proposition).

On the side, it takes a value from some `Show`able type `A`, which it remembers internally (in an enriched variant of the `Result` structure that we saw above) so that it can be collated and displayed at the end.

For example, suppose we measure the `size` of `Trees` like this:

```
Fixpoint size {A} (t : Tree A) : nat :=
  match t with
    | Leaf ⇒ O
    | Node _ l r ⇒ 1 + size l + size r
  end.
```

We can write a dummy property `treeProp` to collect the sizes of the trees we are generating.

```
Definition treeProp (g : nat → G nat → G (Tree nat)) n :=
  forAll (g n (choose (0,n))) (fun t ⇒ collect (size t) true).

(* QuickChick (treeProp genTreeSized 5). *)
```

```
===>
  4947 : 0
  1258 : 1
  673 : 2
  464 : 6
  427 : 5
  393 : 3
  361 : 7
  302 : 4
  296 : 8
  220 : 9
  181 : 10
  127 : 11
  104 : 12
  83 : 13
  64 : 14
  32 : 15
  25 : 16
  16 : 17
  13 : 18
```

```
6 : 19
5 : 20
2 : 21
1 : 23

+++ OK, passed 10000 tests
```

We see that 62.5% of the tests (4947 + 1258 / 10000) are either `Leafs` or empty `Nodes`, while rather few tests have larger sizes.

Compare this with `genTreeSized'`.

```
(* QuickChick (treeProp genTreeSized' 5). *)

===>
  1624 : 0
  571 : 10
  564 : 12
  562 : 11
  559 : 9
  545 : 8
  539 : 14
  534 : 13
  487 : 7
  487 : 15
  437 : 16
  413 : 6
  390 : 17
  337 : 5
  334 : 1
  332 : 18
  286 : 19
  185 : 4
  179 : 20
  179 : 2
  138 : 21
  132 : 3
  87 : 22
  62 : 23
  19 : 24
  10 : 25
  6 : 26
  2 : 27

  +++ OK, passed 10000 tests
```

This generates far fewer tiny examples, likely leading to more efficient testing of interesting properties.

Dealing with Preconditions

A large class of properties that are commonly encountered in property-based testing are *properties with preconditions*. The default QuickChick approach of generating inputs based on type information can be inefficient for such properties, especially for those with sparse preconditions (i.e. ones that are satisfied rarely with respect to their input domain).

Consider a function that inserts a natural number into a sorted list.

```
Fixpoint sorted (l : list nat) :=
  match l with
  | [] ⇒ true
  | x::xs ⇒ match xs with
             | [] ⇒ true
             | y :: ys ⇒ (x <=? y) && (sorted xs)
             end
  end.

Fixpoint insert (x : nat) (l : list nat) :=
  match l with
  | [] ⇒ [x]
  | y::ys ⇒ if x <=? y then x :: l
            else y :: insert x ys
  end.
```

We could test `insert` using the following *conditional* property:

```
Definition insert_spec (x : nat) (l : list nat) :=
  sorted l ==> sorted (insert x l).

(* QuickChick insert_spec. *)

  ===>
    QuickChecking insert_spec
    +++ Passed 10000 tests (17325 discards)
```

To test this property, QuickChick will try to generate random integers `x` and lists `l`, *check* whether the generated `l` is sorted, and, if it is, proceed to check the conclusion. If it is not, it will discard the generated inputs and try again.

As we can see, this can lead to many discarded tests (in this case, about twice as many as successful ones), which wastes a lot of CPU and leads to inefficient testing.

But the wasted effort is the least of our problems! Let's take a peek at the distribution of the lengths of generated lists using `collect`.

```
Definition insert_spec' (x : nat) (l : list nat) :=
  collect (List.length l) (insert_spec x l).

(* QuickChick insert_spec'. *)

  ===>
QuickChecking insert_spec'
3447 : 0
3446 : 1
1929 : 2
```

```
788 : 3
271 : 4
96 : 5
19 : 6
4 : 7
+++ Passed 10000 tests (17263 discards)
```

The vast majority of inputs have length 2 or less!

(This explains something you might have found suspicious in the previous statistics: that 1/3 of the randomly generated lists were already sorted!)

When dealing with properties with preconditions, it is common practice to write custom generators for well-distributed random data that satisfy the property.

For example, we can generate sorted lists with elements between `low` and `high` like this...

```
Fixpoint genSortedList (low high : nat) (size : nat)
             : G (list nat) :=
  match size with
  | O ⇒ ret []
  | S size' ⇒
    if high <? low then
      ret []
    else
      freq [ (1, ret []) ;
             (size, x <- choose (low, high);;
                    xs <- genSortedList x high size';;
                    ret (x :: xs)) ] end.
```

We use a `size` parameter as usual to control the length of generated lists.

If `size` is zero, we can only return the empty list which is always sorted. If `size` is nonzero, we need to perform an additional check whether `high` is less than `low` (in which case we also return the empty list). If it is not, we can proceed to choose to generate a cons cell, with its head generated between `low` and `high` and its tail generated recursively.

```
(* Sample (genSortedList 0 10 10). *)
```

Finally, we can use `forAllShrink` to define a property using the new generator:

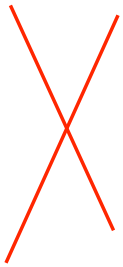
```
Definition insert_spec_sorted (x : nat) :=
  forAllShrink
    (genSortedList 0 10 10)
    shrink
    (fun l ⇒ insert_spec' x l).
```

Now the distribution of lengths looks much better, and we don't discard any tests!

```
QuickChick insert_spec_sorted.

  ===>
    QuickChecking insert_spec_sorted
    947 : 0
    946 : 4
    946 : 10
    938 : 6
```

```
922 : 9
916 : 2
900 : 7
899 : 3
885 : 8
854 : 5
847 : 1
+++ Passed 10000 tests (0 discards)
```

Does this mean we are happy?



Exercise: 5 stars, optional (uniform_sorted)

Using "collect", find out whether generating a sorted list of numbers between 0 and 5 is uniform in the frequencies with which different *numbers* are found in the generated lists.

If not, figure out why. Then write a different generator that achieves a more uniform distribution (preserving uniformity in the lengths).

```
(* FILL IN HERE *)
```

□

Another Precondition: Binary Search Trees

To conclude this chapter, let's look at *binary search trees*.

The `isBST` predicate characterizes trees with elements between `low` and `high`.

```
Fixpoint isBST (low high: nat) (t : Tree nat) :=
  match t with
  | Leaf ⇒ true
  | Node x l r ⇒ (low <? x) && (x <? high)
                 && (isBST low x l) && (isBST x high r)
  end.
```

Here is a (faulty?) insertion function for binary search trees.

```
Fixpoint insertBST (x : nat) (t : Tree nat) :=
  match t with
  | Leaf ⇒ Node x Leaf Leaf
  | Node x' l r ⇒ if x <? x' then Node x' (insertBST x l) r
                  else Node x' l (insertBST x r)
  end.
```

We would expect that if we insert an element that is within the bounds `low` and `high` into a binary search tree, then the result is also a binary search tree.

```
Definition insertBST_spec (low high : nat) (x : nat) (t : Tree nat)
:=
  (low <? x) ==>
  (x <? high) ==>
  (isBST low high t) ==>
  isBST low high (insertBST x t).

(* QuickChick insertBST_spec. *)


 ===>
   QuickChecking insertBST_spec
```

```
0
5
4
Node (4) (Leaf) (Leaf)
*** Failed after 85 tests and 1 shrinks. (1274 discards)
```

We can see that a bug exists when inserting an element into a `Node` with the same payload: if the element already exists in the binary search tree, we should not change it.

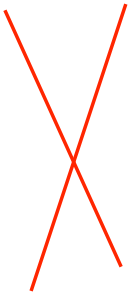
However we are wasting too much testing effort. Indeed, if we fix the bug ...

```
Fixpoint insertBST' (x : nat) (t : Tree nat) :=
  match t with
  | Leaf ⇒ Node x Leaf Leaf
  | Node x' l r ⇒ if x <? x' then Node x' (insertBST' x l) r
                  else if x' <? x then Node x' l (insertBST' x r)
                  else t
  end.

Definition insertBST_spec' (low high : nat) (x : nat) (t : Tree nat)
:=
  (low <? x) ==> (x <? high) ==> (isBST low high t) ==>
  isBST low high (insertBST' x t).
```

... and try again...

```
(* QuickChick insertBST_spec'. *)

===>
QuickChecking insertBST_spec'
*** Gave up! Passed only 1281 tests
Discarded: 20000
```

... we see that 90% of tests are being discarded.



Exercise: 4 stars (gen_bst)

Write a generator that produces binary search trees directly, so that you run 10000 tests with 0 discards.

```
(* FILL IN HERE *)
```

□

```
(* Tue Oct 9 11:47:30 EDT 2018 *)
```