QuickChick Quick Start

Will Thomas

February 9, 2023

1 Introduction

The best way I have found to install QuickChick that minimizes the effort and maintenance is using *opam*. More info can be found at QuickChick Github.

Rather than providing explicit proofs for theorems in Coq, we can test them using QuickChick and then assume that they hold if QuickChick cannot disprove them. This functionality is offered in the form of "Conjectures": A **Conjecture** is the QuickChick equivalent of a normal Coq axiom.

```
Conjecture <NAME> : <PROP>.
```

This is essentially syntactic sugar for:

```
Theorem <NAME> : <PROP>. Admitted.
```

We would then use QuickChick to test our Conjecture via the following command (in Coq):

```
QuickChick <NAME>.
```

There are 4 main ingredients to Property-Based Testing:

1. Executable Properties: We need our properties to be actually testable/executable
2. Generators: We want well defined generators to create random tests
3. Printers: For showing failing examples
4. Shrinkers: To help find the smallest failing examples for easier debugging

2 Typeclasses

Typeclasses are one of the coolest features in Coq. They are roughly equivalent to interfaces where we can know certain things about a type based on its implementing a Typeclass.
This will be a brief introduction to Typeclasses as they have so many features they cannot all be covered here.

There are 2 fundamental parts to Typeclasses:

1. **Class Definitions**: Used to create an interface that Types must instantiate

```
Class <CLASS_NAME> (...ARGS) : <RETURN_TYPE> :=
{
    <FIELD_NAME> : <TYPE_SIGNATURE> ;
}.
```

- Typically the RETURN_TYPE can be omitted (it will typically be inferred properly by Coq)
- You can include any amount of fields in the class definitions

2. **Instantiations**: The actual instantiaion of a Class with arguments and obligations fulfilled

```
<LOCALITY> Instance <INSTANCE_NAME> : <CLASS_NAME> (...ARGS) :=
{
    <FIELD_NAME> : <DEFINITION> ;
}.
```

- The DEFINITION must fulfill the TYPE_SIGNATURE of the corresponding field
- Every field must be instantiated (you cannot leave any fields out)
- It is sometimes helpful to create instances in the *prover* to leverage automation

Example 2.1 The ”Show” Typeclass

```
Class Show A : Type :=
{
    show    : A -> string
}.

Local Instance showBool : Show bool :=
{
    show    := fun b : bool => if b then "true" else "false"
}.
```

or equivalently

```
Local Instance showBool' : Show bool.
constructor; intros b; destruct b eqn:B.
- (* b = true *) apply "true".
- (* b = false *) apply "false".
Defined.
```

We can then use the ”show” function on any type where the *Typeclass Show* has a valid *Instantiation* in the Environment

Let us assume for a moment we have instantiated Show for Nats, bools, and strings (and assume all the obvious implementations for the corresponding ”show” functions). We can then make the following calls

```
Compute (show 21) (* "21" *)
Compute (show true) (* "true" *)
Compute (show "hello") (* "hello" *)
```

This is because the function "show" has been overloaded, and the proper definition (and corresponding Typeclass) will be inferred and applied by Coq.
Other features of Typeclasses to look for (but not to be included here) are:

- Class Hierarchies
- Implicit Generalization
- Typeclass automation tools

3 QuickChick

The Typeclasses that QuickChick uses are

- Show: For printing out the value as a string
- Dec (Decidable Equality): Proving that two values are equal
- Gen (or GenSized): For controlling the generation of a type, requires a definition that outputs the "G" of a type
- Shrink: Helps minimize the size of counter-examples
- Arbitrary: Packages Gen and Shrink Typeclasses together

Generators are the most important feature of property based testing. In QuickChick the generators are somewhat complicated to understand, but fairly easy to use. For that reason, I suggest reading the QC chapter in the QuickChick textbook for a complete understanding; although I will provide a brief overview here.

The **G Monad** is the base of all random generation in QuickChick. A generator for a given type "T" will belong to the type "G T". "G T" is the set of functions that take a random input seed (provided by QuickChick) and output an element of "T". We can build "G T" using a monadic style in Coq that comes as a built in import with QuickChick.

```
(* Roughly copied from copland-avm/CopParserQC.v *)
From QuickChick Require Import QuickChick
Import QcNotation.
Require Export ExtLib.Structures.Monads.
Export MonadNotation.
(* Additional definitions and imports *)

(*** G ASP *)
Definition gASP : G ASP :=
  oneOf [
    (ret NULL) ; (ret CPY) ; (ret SIG) ; (ret HSH) ;
    (par <- arbitrary ;; ret (ASPC ALL EXTD par))
  ].
```

Here we use the monad style to assign/bind ("<-"), sequence (";;"), and return ("ret"). We also use the "arbitrary" Typeclass to generate an arbitrary (or random) set of parameters to ASPC.
We also can use some tools provided by QuickChick to construct **G**:

- oneOf (l : list G A) := returns a single one of the elements from the list. All at the same probability of being returned
- freq (l : list (nat * G A)) := returns a single element from the list, but this time the probability is based upon the number in the pair
(higher number $\implies$ higher probability of being returned)
- choose (p : (nat * nat)) := returns a random number in between the lower bound and upper bound given by the pair (fst = lower, snd = upper).

Others also exist, but I have not had to use them so they are omitted (but available in the QuickChick textbook > QuickChickInterface chapter).
Power Tools:
The most powerful feature of QuickChick is the automatic derivation of Typeclasses for many Types. This can be leveraged by using

```
Derive <TYPECLASS> for <TYPE>.
```

Which will automatically create the given Typeclass for the Type if possible. You can use derive with the following options (and more less commonly used ones): Show, Shrink, Arbitrary.

4 Summary and Use

Hopefully the benefits of QuickChick are clear. We can easily check propositions to make sure they do not have obvious counter-examples. From there, we diagnose the problem: is this a case of a poorly constructed definition or an actually false proposition? Iterate and repeat until QuickChick can find no counter-examples, then attempt a proof.

The most foolproof way I have found of doing this is to start with the smallest structure that will be used within your proposition, create the necessary Typeclasses, test the Typeclasses, then move up one level and repeat.
For example if we had the following (buggy) spec:

```
Inductive Tree (A : Type) :=
| Leaf : Tree A
| Node : A -> Tree A -> Tree A -> Tree A.
Arguments Leaf {A}.
Arguments Node {A} _ _ _.

Inductive Plc :=
| Plc_Str : string -> Plc
| Plc_nat : nat -> Plc.

Fixpoint longest_depth {A : Type} (t : Tree A) : nat :=
  match t with
  | Leaf => 0
  | Node a tl tr =>
      let l_depth := longest_depth tl in
      let r_depth := longest_depth tr in
      if (Nat.leb l_depth r_depth) then r_depth else l_depth
  end.

Conjecture all_trees_depth : forall (p : Plc) (tl tr : Tree Plc),
  longest_depth tl < longest_depth (Node p tl tr).
```

The steps we would have to take before we can actually ”QuickChick” our Conjecture would be to instantiate the following (typically in order):

1. Show

2. Dec (Decidable Equality)

3. G

4. Gen (or GenSized)

5. Shrink

6. Arbitrary

And do this for each of the following Types (also usually in order):

1. ascii

2. String

3. Plc

4. Tree

Then we can execute

```
QuickChick all_trees_depth
(*
  QuickChecking all_trees_depth
  Plc_nat 0
  Leaf
  Leaf
  *** Failed after 1 tests and 0 shrinks. (0 discards)
*)
```

Looking back at our definition, we notice that ”longest_depth” is incorrectly defined. With this fix we will succeed

```
Fixpoint longest_depth {A : Type} (t : Tree A) : nat :=
  match t with
  | Leaf => 0
  | Node a tl tr =>
      let l_depth := longest_depth tl in
      let r_depth := longest_depth tr in
      if (Nat.leb l_depth r_depth) then 1 + r_depth else 1 + l_depth
```

```
    end.

  QuickChick all_trees_depth
  (* +++ Passed 10000 tests (0 discards) *)
```

Important Notes:

- QuickChick will only be as powerful as the generator you create. If your generators never create random corner cases, then QuickChick will never catch them.
- If you use automated derivation (via ”Derive” commands), make sure you test that the derivation makes sense and has not just created an overly simplified Typeclass instance.
- I personally believe ”Shrink” is the hardest Typeclass to create (and also has certain bugs in QuickChick related to it). If you are struggling to create a Shrink, use the trivial Shrink and move on.
- ”Set Typeclasses Debug” is a very helpful command to locate what Typeclasses cannot be found when you cannot execute the ”QuickChick” command.