SOFTWARE FOUNDATIONS

VOLUME 4: QUICKCHICK: PROPERTY-BASED TESTING IN COQ

TABLE OF CONTENTS INDEX

QUICKCHICKINTERFACE

QUICKCHICK REFERENCE MANUAL

```
From QuickChick Require Import QuickChick.
Require Import ZArith Strings.Ascii Strings.String.

From ExtLib.Structures Require Import Functor Applicative.
```

QuickChick provides a large collection of combinators and notations for writing property-based random tests. This file documents the entire public interface (the module type `QuickChickSig`).

```
Module Type QuickChickSig.
```

The `Show` Typeclass

`Show` typeclass allows the test case to be printed as a string.

```
  Class Show (A : Type) : Type :=
    {
      show : A → string
    }.
```

Here are some `Show` instances for some basic types:

```
Declare Instance showNat : Show nat.
Declare Instance showBool : Show bool.
Declare Instance showZ : Show Z.
Declare Instance showString : Show string.

Declare Instance showList :
  ∀ {A : Type} `{Show A}, Show (list A).
Declare Instance showPair :
  ∀ {A B : Type} `{Show A} `{Show B}, Show (A * B).
Declare Instance showOpt :
  ∀ {A : Type} `{Show A}, Show (option A).
```

```
Declare Instance showEx :
  ∀ {A} `{Show A} P, Show ({x : A | P x}).
```

When defining `Show` instance for your own datatypes, you sometimes need to start a new line for better printing. `nl` is a shorthand for it.

```
Definition nl : string := String (ascii_of_nat 10) EmptyString.
```

Generators

Fundamental Types

A `RandomSeed` represents a particular starting point in a pseudo-random sequence.

```
Parameter RandomSeed : Type.
```

`G A` is the type of random generators for type `A`.

```
Parameter G : Type → Type.
```

Run a generator with a size parameter (a natural number denoting the maximum depth of the generated A) and a random seed.

```
Parameter run : ∀ {A : Type}, G A → nat → RandomSeed → A.
```

The semantics of a generator is its set of possible outcomes.

```
Parameter semGen : ∀ {A : Type} (g : G A), set A.
Parameter semGenSize : ∀ {A : Type} (g : G A) (size : nat), set
A.
```

Structural Combinators

Generators are also instances of several generic typeclasses. Many handy generator combinators can be found in the `Monad`, `Functor`, `Applicative`, `Foldable`, and `Traversable` modules in the `ExtLib.Structures` library from `coq-ext-lib`.

```
Declare Instance Monad_G : Monad G.
Declare Instance Functor_G : Functor G.
Declare Instance Applicative_G : Applicative G.
```

A variant of monadic bind where the continuation also takes a *proof* that the value received is within the set of outcomes of the first generator.

```
Parameter bindGen' : ∀ {A B : Type} (g : G A),
    (∀ (a : A), (a ∈ semGen g) → G B) → G B.
```

A variant of bind for the `(G (option --))` monad. Useful for chaining generators that can fail / backtrack.

```
Parameter bindGenOpt : ∀ {A B : Type},
    G (option A) → (A → G (option B)) → G (option B).
```

Basic Generator Combinators

The `listOf` and `vectorOf` combinators construct generators for `list A`, provided a generator `g` for type `A`: `listOf g` yields an arbitrary-sized list (which might be empty), while `vectorOf n g` yields a list of fixed size `n`.

```
Parameter listOf : ∀ {A : Type}, G A → G (list A).
Parameter vectorOf : ∀ {A : Type}, nat → G A → G (list A).
```

`elems_ a l` constructs a generator from a list `l` and a default element `a`. If `l` is non-empty, the generator picks an element from `l` uniformly; otherwise it always yields `a`.

```
Parameter elems_ : ∀ {A : Type}, A → list A → G A.
```

Similar to `elems_`, instead of choosing from a list of `A`s, `oneOf_ g l` returns `g` if `l` is empty; otherwise it uniformly picks a generator for `A` in `l`.

```
Parameter oneOf_ : ∀ {A : Type}, G A → list (G A) → G A.
```

We can also choose generators with distributions other than the uniform one. `freq_ g l` returns `g` if `l` is empty; otherwise it chooses a generator from `l`, where the first field indicates the chance that the second field is chosen. For example, `freq_ z [(2, x); (3, y)]` has 40% probability of choosing `x` and 60% probability of choosing `y`.

```
Parameter freq_ :
  ∀ {A : Type}, G A → list (nat * G A) → G A.
```

Try all generators until one returns a `Some` value or all failed once with `None`. The generators are picked at random according to their weights (like `frequency`), and each one is run at most once.

```
Parameter backtrack :
  ∀ {A : Type}, list (nat * G (option A)) → G (option A).
```

Internally, the G monad hides a `size` parameter that can be accessed by generators. The `sized` combinator provides such access. The `resize` combinator sets it.

```
Parameter sized : ∀ {A: Type}, (nat → G A) → G A.
Parameter resize : ∀ {A: Type}, nat → G A → G A.
```

Generate-and-test approach to generate data with preconditions.

```
Parameter suchThatMaybe :
  ∀ {A : Type}, G A → (A → bool) → G (option A).
Parameter suchThatMaybeOpt :
  ∀ {A : Type}, G (option A) → (A → bool) → G (option A).
```

The `elems_`, `oneOf_`, and `freq_` combinators all take default values; these are only used if their list arguments are empty, which should not normally happen. The `QcDefaultNotation` sub-module exposes notation (without the underscores) to hide this default.

```
Module QcDefaultNotation.
```

elems is a shorthand for elems_ without a default argument.

```
  Notation " 'elems' [ x ] " :=
    (elems_ x (cons x nil)) : qc_scope.
  Notation " 'elems' [ x ; y ] " :=
    (elems_ x (cons x (cons y nil))) : qc_scope.
  Notation " 'elems' [ x ; y ; .. ; z ] " :=
    (elems_ x (cons x (cons y .. (cons z nil) ..))) : qc_scope.
  Notation " 'elems' ( x ;; l ) " :=
    (elems_ x (cons x l)) (at level 1, no associativity) :
qc_scope.
```

oneOf is a shorthand for oneOf_ without a default argument.

```
  Notation " 'oneOf' [ x ] " :=
    (oneOf_ x (cons x nil)) : qc_scope.
  Notation " 'oneOf' [ x ; y ] " :=
    (oneOf_ x (cons x (cons y nil))) : qc_scope.
  Notation " 'oneOf' [ x ; y ; .. ; z ] " :=
    (oneOf_ x (cons x (cons y .. (cons z nil) ..))) : qc_scope.
  Notation " 'oneOf' ( x ;; l ) " :=
    (oneOf_ x (cons x l)) (at level 1, no associativity) :
qc_scope.
```

freq is a shorthand for freq_ without a default argument.

```
  Notation " 'freq' [ x ] " :=
    (freq_ x (cons x nil)) : qc_scope.
  Notation " 'freq' [ ( n , x ) ; y ] " :=
    (freq_ x (cons (n, x) (cons y nil))) : qc_scope.
  Notation " 'freq' [ ( n , x ) ; y ; .. ; z ] " :=
    (freq_ x (cons (n, x) (cons y .. (cons z nil) ..))) :
qc_scope.
  Notation " 'freq' ( ( n , x ) ;; l ) " :=
    (freq_ x (cons (n, x) l)) (at level 1, no associativity) :
qc_scope.

End QcDefaultNotation.
```

The original version of QuickChick used elements, oneof and frequency as the default-argument versions of the corresponding combinators. These have since been deprecated in favor of a more consistent naming scheme.

Choosing from Intervals

The combinators above allow us to generate elements by enumeration and lifting. However, for numeric data types, we sometimes hope to choose from an interval without writing down all the possible values.

Such intervals can be defined on ordered data types, instances of OrdType, whose ordering leq satisfies reflexive, transitive, and antisymmetric predicates.

```
Existing Class OrdType.
```

```
Declare Instance OrdBool : OrdType bool.
Declare Instance OrdNat : OrdType nat.
Declare Instance OrdZ : OrdType Z.
```

We also expect the random function to be able to pick every element in any given interval.

```
Existing Class ChoosableFromInterval.
```

QuickChick has provided some instances for ordered data types that are choosable from intervals, including `bool`, `nat`, and `Z`.

```
Declare Instance ChooseBool : ChoosableFromInterval bool.
Declare Instance ChooseNat : ChoosableFromInterval nat.
Declare Instance ChooseZ : ChoosableFromInterval Z.
```

`choose l r` generates a value between `l` and `r`, inclusive the two extremes. It causes a runtime error if `r` < `l`.

```
Parameter choose :
  ∀ {A : Type} `{ChoosableFromInterval A}, (A * A) → G A.
```

The `Gen` and `GenSized` Typeclasses

`GenSized` and `Gen` are typeclasses whose instances can be generated randomly. More specifically, `GenSized` depends on a generator for any given natural number that indicate the size of output.

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

Given an instance of `GenSized`, we can convert it to `Gen` automatically, using `sized` function.

```
Declare Instance GenOfGenSized {A} `{GenSized A} : Gen A.
```

Here are some basic instances for generators:

```
Declare Instance genBoolSized : GenSized bool.
Declare Instance genNatSized : GenSized nat.
Declare Instance genZSized : GenSized Z.

Declare Instance genListSized :
  ∀ {A : Type} `{GenSized A}, GenSized (list A).
Declare Instance genList :
  ∀ {A : Type} `{Gen A}, Gen (list A).
Declare Instance genOption :
  ∀ {A : Type} `{Gen A}, Gen (option A).
Declare Instance genPairSized :
  ∀ {A B : Type} `{GenSized A} `{GenSized B}, GenSized (A*B).
Declare Instance genPair :
  ∀ {A B : Type} `{Gen A} `{Gen B}, Gen (A * B).
```

Generators for Data Satisfying Inductive Predicates

Just as QuickChick provides the `GenSized` and `Gen` typeclasses for generators of type `A`, it provides constrained variants for generators of type `A` such that `P : A → Prop` holds of all generated values. Since it is not guaranteed that any such `A` exist, these generators are partial.

```
Class GenSizedSuchThat (A : Type) (P : A → Prop) :=
  {
    arbitrarySizeST : nat → G (option A)
  }.

Class GenSuchThat (A : Type) (P : A → Prop) :=
  {
    arbitraryST : G (option A)
  }.
```

So, for example, if you have a typing relation `has_type : exp → type → Prop` for some language, you could, given some type `T` as input, write (or derive as we will see later on) an instance of `GenSizedSuchThat (fun e ⇒ has_type e T)`, that produces an expression of with type `T`.

Calling `arbitraryST` through such an instance would require making an explicit application to `@arbitraryST` as follows:

```
@arbitraryST _ (fun e ⇒ has_type e T) _
```

where the first placeholder is the type of expressions `exp` and the second placeholder is the actual instance to be inferred.

To avoid this, QuickChick also provides convenient notation to call by providing only the predicate `P` that constraints the generation. The typeclass constraint is inferred.

```
Notation "'genST' x" := (@arbitraryST _ x _) (at level 70).
```

Shrinking

The **`Shrink`** Typeclass

`Shrink` is a typeclass whose instances have an operation for shrinking larger elements to smaller ones, allowing QuickChick to search for a minimal counter example when errors occur.

```
Class Shrink (A : Type) :=
  {
    shrink : A → list A
  }.
```

Default shrinkers for some basic datatypes:

```
Declare Instance shrinkBool : Shrink bool.
Declare Instance shrinkNat : Shrink nat.
Declare Instance shrinkZ : Shrink Z.

Declare Instance shrinkList {A : Type} `{Shrink A} : Shrink
(list A).
Declare Instance shrinkPair {A B} `{Shrink A} `{Shrink B} :
Shrink (A * B).
Declare Instance shrinkOption {A : Type} `{Shrink A} : Shrink
(option A).
```

The `Arbitrary` Typeclass

The `Arbitrary` typeclass combines generation and shrinking.

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

The Generator Typeclass Hierarchy

```
            GenSized
               |
               |
              Gen    Shrink
                \     /
                 \   /
               Arbitrary
```

If a type has a `Gen` and a `Shrink` instance, it automatically gets an `Arbitrary` one.

```
Declare Instance ArbitraryOfGenShrink :
  ∀ {A} `{Gen A} `{Shrink A}, Arbitrary A.
```

Checkers

Basic Definitions

`Checker` is the opaque type of QuickChick properties.

```
Parameter Checker : Type.
```

The `Checkable` class indicates we can check a type A.

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

Boolean checkers always pass or always fail.

```
Declare Instance testBool : Checkable bool.
```

The unit checker is always discarded (that is, it represents a useless test). It is used, for example, in the implementation of the "implication `Checker`" combinator `==>`.

```
Declare Instance testUnit : Checkable unit.
```

Given a generator for showable `A`s, construct a `Checker`.

```
Parameter forAll :
  ∀ {A prop : Type} `{Checkable prop} `{Show A}
         (gen : G A) (pf : A → prop), Checker.
```

A variant of `forAll` that provides evidence that the generated values are members of the semantics of the generator. (Such evidence can be useful when constructing dependently typed data, such as bounded integers.)

```
Parameter forAllProof :
  ∀ {A prop : Type} `{Checkable prop} `{Show A}
         (gen : G A) (pf : ∀ (x : A), semGen gen x → prop),
Checker.
```

Given a generator and a shrinker for showable `A`s, construct a `Checker`.

```
Parameter forAllShrink :
  ∀ {A prop : Type} `{Checkable prop} `{Show A}
         (gen : G A) (shrinker : A → list A) (pf : A → prop),
Checker.
```

Lift (`Show`, `Gen`, `Shrink`) instances for `A` to a `Checker` for functions `A` -> prop. This is what makes it possible to write (for some example property `foo := fun x ⇒ x >? 0`, say) `QuickChick foo` instead of `QuickChick (forAllShrink arbitrary shrink foo)`.

```
Declare Instance testFun :
  ∀ {A prop : Type} `{Show A} `{Arbitrary A} `{Checkable prop},
    Checkable (A → prop).
```

Lift products similarly.

```
Declare Instance testProd :
  ∀ {A : Type} {prop : A → Type} `{Show A} `{Arbitrary A}
         `{∀ x : A, Checkable (prop x)},
    Checkable (∀ (x : A), prop x).
```

Lift polymorphic functions by instantiating to 'nat'. :-)

```
Declare Instance testPolyFun :
  ∀ {prop : Type → Type} `{Checkable (prop nat)},
    Checkable (∀ T, prop T).
```

Checker Combinators

Print a specific string if the property fails.

```
Parameter whenFail :
  ∀ {prop : Type} `{Checkable prop} (str : string), prop →
  Checker.
```

Record an expectation that a property should fail, i.e. the property will fail if all the tests succeed.

```
  Parameter expectFailure :
    ∀ {prop: Type} `{Checkable prop} (p: prop), Checker.
```

Collect statistics across all tests.

```
  Parameter collect :
    ∀ {A prop : Type} `{Show A} `{Checkable prop} (x : A),
      prop → Checker.
```

Set the reason for failure. Will only count shrinks as valid if they preserve the tag.

```
  Parameter tag :
    ∀ {prop : Type} `{Checkable prop} (t : string), prop →
  Checker.
```

Form the conjunction / disjunction of a list of checkers.

```
  Parameter conjoin : ∀ (l : list Checker), Checker.
  Parameter disjoin : ∀ (l : list Checker), Checker.
```

Define a checker for a conditional property. Invalid generated inputs (ones for which the antecedent fails) are discarded.

```
  Parameter implication :
    ∀ {prop : Type} `{Checkable prop} (b : bool) (p : prop),
  Checker.
```

Notation for implication. Clashes with many other notations in other libraries, so it lives in its own module. Note that this includes the notations for the generator combinators above to avoid needing to import two modules.

```
  Module QcNotation.
    Export QcDefaultNotation.

    Notation "x ==> y" :=
      (implication x y) (at level 55, right associativity)
      : Checker_scope.
  End QcNotation.
```

Decidability

The Dec Typeclass

Decidability typeclass using ssreflect's 'decidable'.

```
      Class Dec (P : Prop) : Type := { dec : decidable P }.
```

Decidable properties are Checkable.

```
  Declare Instance testDec {P} `{H : Dec P} : Checkable P.
```

Logic Combinator instances.

```
Declare Instance Dec_neg {P} {H : Dec P} : Dec (¬ P).
Declare Instance Dec_conj {P Q} {H : Dec P} {I : Dec Q} : Dec (P
∧ Q).
Declare Instance Dec_disj {P Q} {H : Dec P} {I : Dec Q} : Dec (P
∨ Q).
```

A convenient notation for coercing a decidable proposition to a `bool`.

```
Notation "P '?'" := (match (@dec P _) with
                     | left _ ⇒ true
                     | right _ ⇒ false
                     end) (at level 100).
```

The `Dec_Eq` Typeclass

```
   Class Dec_Eq (A : Type) :=
     {
       dec_eq : ∀ (x y : A), decidable (x = y)
     }.
```

Automation and conversions for Dec.

```
Declare Instance Eq__Dec {A} `{H : Dec_Eq A} (x y : A) : Dec (x
= y).
```

Since deciding equalities is a very common requirement in testing, QuickChick provides a tactic that can define instances of the form `Dec (x = y)`.

```
   Ltac dec_eq.
```

QuickChick also lifts common decidable instances to the `Dec` typeclass.

```
Declare Instance Dec_eq_bool (x y : bool) : Dec (x = y).
Declare Instance Dec_eq_nat (m n : nat) : Dec (m = n).
Declare Instance Dec_eq_opt (A : Type) (m n : option A)
`{_ : ∀ (x y : A), Dec (x = y)} : Dec (m = n).
Declare Instance Dec_eq_prod (A B : Type) (m n : A * B)
`{_ : ∀ (x y : A), Dec (x = y)}
`{_ : ∀ (x y : B), Dec (x = y)}
: Dec (m = n).
Declare Instance Dec_eq_list (A : Type) (m n : list A)
`{_ : ∀ (x y : A), Dec (x = y)} : Dec (m = n).

Declare Instance Dec_ascii (m n : Ascii.ascii) : Dec (m = n).
Declare Instance Dec_string (m n : string) : Dec (m = n).
```

Automatic Instance Derivation

QuickChick allows the automatic derivation of typeclass instances for simple types:

```
        Derive <class> for T.
```

Here `<class>` must be one of `GenSized`, `Shrink`, `Arbitrary`, or `Show`, and `T` must be an inductive defined datatype (think Haskell/OCaml).

To derive multiple classes at once, write:

```
        Derive (<class>,...,<class>) for T.
```

QuickChick also allows for the automatic derivation of generators satisfying preconditions in the form of inductive relations:

Derive ArbitrarySizedSuchThat for (fun x => P x_1 ... x .... xn).

<P> must be an inductively defined relation. <x> is the function to be generated. <x_1...xn> are (implicitly universally quantified) variable names.

QuickChick also allows automatic derivations of proofs of correctness of its derived generators! For more, look at:

- A paper on deriving QuickChick generators for a large class of inductive relations. http://www.cis.upenn.edu/~llamp/pdf/GeneratingGoodGenerators.pdf
- Leo's PhD dissertation. https://lemonidas.github.io/pdf/Leo-PhD-Thesis.pdf
- examples/DependentTest.v

Top-level Commands and Settings

QuickChick provides a series of toplevel commands to sample generators, test properties, and derive useful typeclass instances.

The `Sample` command samples a generator. The argument `g` needs to have type `G A` for some showable type `A`.

```
    Sample g.
```

The main testing command, `QuickChick`, runs a test. The argument `prop` must belong to a type that is an instance of `Checkable`.

```
     QuickChick prop.
```

QuickChick uses arguments to customize execution.

```
  Record Args :=
    MkArgs
      {
        (* Re-execute a test. *)
        (* Default: None *)
        replay : option (RandomSeed * nat);
        (* Maximum number of successful tests to run. *)
        (* Default: 10000 *)
        maxSuccess : nat;
        (* Maximum number of discards to accept. *)
        (* Default: 20000 *)
        maxDiscard : nat;
        (* Maximum number of shrinks to perform before terminating. *)
        (* Default : 1000 *)
        maxShrinks : nat;
        (* Maximum size of terms to generate (depth). *)
        (* Default : 7 *)
        maxSize : nat;
        (* Verbosity. Note: Doesn't do much... *)
        (* Default true. *)
        chatty : bool
      }.
```

Instead of record updates, you should overwrite extraction constants.

```
Extract Constant defNumTests    ⇒ "10000".
Extract Constant defNumDiscards ⇒ "(2 * defNumTests)".
Extract Constant defNumShrinks  ⇒ "1000".
Extract Constant defSize        ⇒ "7".
```

The `quickChick` Command-Line Tool

QuickChick comes with a command-line tool that supports:

- Batch processing, compilation and execution of tests
- Mutation testing
- Sectioning of tests and mutants

Comments that begin with an exclamation mark are special to the QuickChick command-line tool parser and signify a test, a section, or a mutant.

Test Annotations

A test annotation is just a `QuickChick` command wrapped inside a comment with an exclamation mark.

```
(*! QuickChick prop. *)
```

Only tests that are annotated this way will be processed. Only property names are allowed.

Mutant Annotations

A mutant annotation consists of 4 components. First an anottation that signifies the beginning of the mutant `! *)`. That is followed by the actual code. Then, we can include an optional annotation (in a comment with double exclamation marks) that corresponds to the mutant names. Finally, we can add a list of mutations inside normal annotated comments. Each mutant should be able to be syntactically substituted in for the normal code.

```
(*! *)
Normal code
(*!! mutant-name *)
(*! mutant 1 *)
(*! mutant 2 *)
... etc ...
```

Section Annotations

To organize larger developments better, we can group together different tests and mutants in sections. A section annotation is a single annotation that defines the beginning of the section (which lasts until the next section or the end of the file).

```
(*! Section section-name *)
```

Optionally, one can include an extends clause

```
(*! Section section-name *)(*! extends other-section-name *)
```

This signifies that the section being defined also contains all tests and mutants from `other-section-name`.

Command-Line Tool Flags

The QuickChick command line tool can be passed the following options:

- `-s <section>`: Specify which sections properties and mutants to test
- `-v`: Verbose mode for debugging
- `-failfast`: Stop as soon as a problem is detected
- `-color`: Use colors on an ANSI-compatible terminal
- `-cmd <command>`: What compile command is used to compile the current directory if it is not `make`
- `-top <name>`: Specify the name of the top-level logical module. That should be the same as the `-Q` or `-R` directive in `_CoqProject` or `Top` which is the default
- `-ocamlbuild <args>`: Any arguments necessary to pass to ocamlbuild when compiling the extracted code (e.g. linked libraries)
- `-nobase`: Pass this option to not test the base mutant
- `-m <number>`: Pass this to only test a mutant with a specific id number
- `-tag <name>`: Pass this to only test a mutant with a specific tag
- `-include <name>`: Specify a to include in the compilation
- `-exclude <names>`: Specify files to be excluded from compilation. Must be the last argument passed.

Deprecated Features

The following features are retained for backward compatibility, but their use is deprecated.

Use the monad notations from `coq-ext-lib` instead of the `QcDoNotation` sub-module:

```
Module QcDoNotation.
  Notation "'do!' X <- A ; B" :=
    (bindGen A (fun X ⇒ B))
    (at level 200, X ident, A at level 100, B at level 200).
  Notation "'do\'' X <- A ; B" :=
    (bindGen' A (fun X H ⇒ B))
    (at level 200, X ident, A at level 100, B at level 200).
  Notation "'doM!' X <- A ; B" :=
    (bindGenOpt A (fun X ⇒ B))
    (at level 200, X ident, A at level 100, B at level 200).
End QcDoNotation.

End QuickChickSig.
```