A Coq Framework For Verified Property-Based Testing

Internship Report

Inria Paris-Rocquencourt

Zoe Paraskevopoulou (Intern)

Cătălin Hriţcu (Supervisor)

initial version: September 18, 2014

last revised: October 17, 2014

Abstract

While random property-based testing is often an effective way for quickly finding bugs and increasing software quality, testing errors can conceal important bugs and thus reduce its benefits. In this report we introduce a novel methodology for formally verified property-based testing; this methodology is embodied by a framework built on top of the QuickChick testing plugin for Coq. Our verification framework is aimed at proving the correctness of executable testing code with respect to a high-level specification, which captures the conjecture under test in a more direct way. To this end, we provide a systematic way for reasoning about the set of values a random data generator can produce with non-zero probability. We have used our methodology to prove the correctness of most QuickChick combinators, with respect to the axiomatic semantics of a small number of primitive ones. We have also applied our methodology on a red-black tree example and made good progress on a more complex noninterference example. These encouraging preliminary results indicate that this verification methodology is modular, scalable, and requires minimal changes to existing code.

Contents

Chapter 1

Introduction

Software testing is a widely accepted method of software validation, aiming to check whether the software under test meets its requirements or not. Although, as Dijkstra famously quoted, testing can only show the presence of bugs, in practice, it is the most commonly used approach for ensuring the quality of software.

Randomized property-based testing has gained a lot of popularity since the QuickCheck project started in 1999 [7, 13], especially in the functional programming community, giving the ability to quickly test formal specifications of programs on a large number of randomly generated inputs. The programmer writes *checkers*, executable programs that test the desired specification. Checkers are written in the same language as the programs under test. If testing fails, a counterexample is returned, providing useful information for debugging. The programmer may also need to write random *generators* in order to generate user defined data types or fine-tune a generator that is being used by a specific checker.

However, one may wonder, "who watches the watchmen?"; how much confidence can we have about the program under test adhering to its specifications, when the testing cannot find any more bugs? In fact there are quite a few things that can tamper with the effectiveness of property-based testing. Two common causes of ineffective testing are bugs in the test data generators and bugs in checkers. There can be various types of bugs in the generators: a generator may fail to cover sufficiently the input space or may lead to a lot of discarded test cases for which the specification under test holds vacuously as they fail to satisfy a certain precondition. On the other hand, checkers may fail to capture the desired high-level specification, especially when it comes to large systems with complex invariants.

Several methods have been proposed in order to evaluate the effectiveness of testing such as coverage metrics based on control or data flow [20, 9], mutation testing [14, 15] etc. The goal of this report is to propose a way to gain formal guarantees about the quality of testing by proving that the conjecture we are checking corresponds to a high-level declarative specification. Therefore, the programmer need not trust that the checker captures the intended conjecture but only trust the high-level declarative specification corresponds to the desired requirement, which is generally the case in formal verification. We devise a mechanism to automatically map both generators and checkers to declarative semantics that we can be used to prove them equivalent to the desired high-level declarative specifications. The guarantee that this method provides is that if we could enumerate the output space of the generators used without producing any counter-examples then we would have a proof by exhaustion for the desired declarative specification. While exhaustion is very rarely possible in practice and would be incompatible with our randomized approach, this theoretical guarantee ensures us that we are thoroughly testing the correct conjecture. Most bugs in real generators and checkers are breaking even this rather weak guarantee. This verification methodology is demonstrated in fig. 1.1.

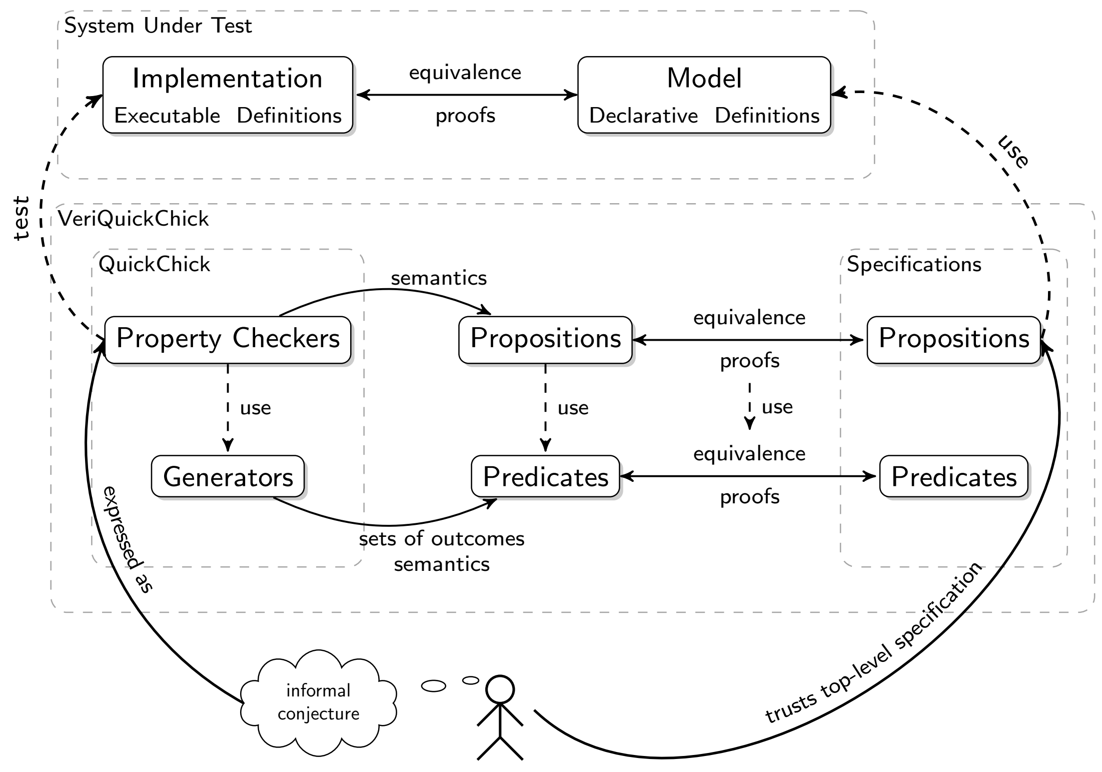

Figure 1.1: The proposed verification methodology

Our framework is built on top of QuickChick,[1] a Coq clone of QuickCheck for Haskell [7, 13]. Our extension allows us to reason about generators by mapping each generator to its set of outcomes, the set of values that have non-zero probability of being generated. Given the set of outcomes of a generator we can prove that the generator is *sound* (i.e. all the possible outcomes satisfy a predicate) and *complete* (i.e. all the values that satisfy a predicate can be generated) with respect to a higher-level specification (also a set of outcomes). By using the set of outcomes semantics for generators we can automatically map each checker to a logical proposition in Coq. We can then prove that proposition equivalent to a high-level specification (another Coq proposition) that captures the conjecture being tested. This proof of course only succeeds if the checker is correct and indeed tests to the right conjecture.

As in QuickCheck, in QuickChick one can use a large library of reusable generator combinators, in order to write custom generators. We want our extension to be seamlessly integrated into QuickChick, without requiring any significant changes to existing code. We achieve this by overloading each generator combinator with a dual semantics. We can either map a generator to the standard semantics and run it for a given random seed in order to obtain input data, or map it to its set of possible outcomes in order to reason about its output space. In order to avoid proof duplication when proving properties about the set of outcomes of generators, we prove each generator combinator in the QuickChick library correct with respect to a high-level specification, producing a set of reusable lemmas. This way we can structure the proofs for the generators that use the combinators in a compositional way, achieving a high level of reusability

[1]The QuickChick development, along with our fully-integrated verification framework, can be found at `https://github.com/QuickChick/QuickChick`

and independence from the implementation of each combinator.

Similarly to generators, the user can use combinators from the QuickChick library to write checkers. The simplest way of writing a checker is writing a boolean predicate and test it with input produced by generators associated to the input types. We can trivially map a boolean predicate to a logical predicate by requiring the result be `true` for each and every possible input that belongs to the set of outcomes of the generators being used. However, the user can form more complex checkers by using a series of combinators to fine-tune the testing process by specifying the generators to be used, changing the expected testing result, collecting information about test case distribution, etc. As with generator combinators, we can give checker combinators dual semantics, hence they can be used either for the actual testing or for obtaining logical propositions. As with generators, we provide a series of reusable lemmas for the checker combinators in QuickChick in order to make the proving process of user-level checkers easier.

Nonetheless, even with the maximum amount of reusability, formal verification requires a considerable amount of human effort and our verification framework requires the user to do manual proofs. One of the future directions of QuickChick includes the development of a framework for automatically deriving generators from formal specifications. Our verification framework could be used in verifying such a generation framework, achieving both automation and formal guarantees.

The aim of this report is to present in detail our verification methodology and evaluate it through examples and a larger case study.

Report Outline

The rest of the report is organized as follows: In chapter 2 we briefly present randomized property-based testing in general and how testing and proving can be combined. In chapter 3 we introduce the QuickChick framework and we describe how it works. In chapter 4, the main chapter of this report, we present the extensions we made to the tool in order to provide a generic framework for formally verifying testing code. In chapter 5 we evaluate our methodology through a red-black tree case study and we describe our experiences from applying our methodology to the infrastructure used to test an information control-flow machine. In the final chapters we discuss related work as well as conclusions and future work.

Chapter 2

Randomized Property-Based Testing in Coq

2.1 Randomized Property-Based Testing

Randomized property-based testing (RPBT) has gained a lot of popularity since the appearance of QuickCheck for Haskell [7], and has been re-implemented in numerous other languages, including Erlang [18, 3], Prolog [1] and even proof assistants like Isablell/HOL [4, 5] and Agda [8]. RPBT is a form of black-box testing in which a property of a program is tested, by expressing the property as an executable checker, and making sure that this checker succeeds for a large number of randomly generated inputs. If the property is falsified, then a counterexample is returned, often shrinked in order to be more useful in the debugging procedure.

RPBT achieves a high level of automation by combining two ideas, executable properties as oracles and random input data generation. This means that the user need not to check each output separately nor write test data manually, as is commonly the case with more traditional unit testing frameworks in which the user must provide both the test cases and their corresponding expected outputs.

However, testing complex properties may need fine-tuned custom generators in order for testing to be efficient. For instance, conditional properties, i.e. properties that require the input to satisfy a certain precondition, are hard test effectively when the precondition is satisfied only by a very small proportion of the possible inputs. In such cases, writing custom generators may greatly improve the efficacy of testing, as this allows the user to have control over the probability distribution of the test cases. In addition, custom generators sometimes need to be provided for generation of user defined data types.

2.2 The Value of Counterexamples or Lack Thereof

RPBT facilitates the design and debugging process by providing counterexamples to failing properties. Such counterexamples can be very useful in understanding and fixing bugs in both programs and properties. However, failure reports can be long and it may take manual effort to extract the useful information that reveals the bug. In fact, when it comes to randomly generated inputs the failing case may contain a considerable amount of irrelevant “noise”. QuickCheck and QuickCheck-inspired tools provide a shrinking mechanism which tries to isolate the part of the failing input that triggers the failure by repeatedly simplifying the counterexample until any further simplification does not trigger the failure. Although this mechanism provides no formal guarantees about the minimality of the returned counterexample, it greatly reduces the time needed to locate the fault [13].

When a counterexample is encountered it means that the conjecture being tested is wrong. However, a counterexample can be spurious in the sense that it lies outside the set of values for which the specification is supposed to hold. In that case, one may need to reformulate the checkers that test the specification in order to restrict their domain by adding or strengthening preconditions.

Nevertheless, the inability of QuickCheck to produce counterexamples does not imply that the program under test is correct. In fact, depending on the quality of our testing infrastructure, testing can miss obvious bugs and thus provide very little assurance about the validity of our property. One reason testing could fail to find counterexamples to wrong conjectures is the insufficient coverage of the state space of the property. A generator that only covers a subset of the valid inputs may miss counterexamples that have no chance of being generated. The distribution of the test data also affects the effectiveness of testing. Even if there is non-zero probability for all the valid inputs to be randomly generated the underlying probability distribution may be very biased towards some values making it very hard to hit counterexamples that have an extremely small probability of being generated.

Generators can also fail to satisfy sparse preconditions of conditional properties, leading to a large number of discarded inputs. This can greatly affect the efficiency of testing as a large portion of the generated data may be discards causing interesting test cases to be less frequent and thus having lower probability of hitting counterexamples.

Checkers can also be the reason of inadequate testing as they are executable programs and like all programs may contain bugs. Turning high-level specifications to efficiently executable properties can be an error-prone task. In addition, it is a common practice to use other programs to classify the output and, unless they are formally verified, this programs may not behave correctly. In such cases testing can easily miss counterexamples as, for instance, an erroneously strong precondition will discard test cases for which the property should actually be tested.

2.3 Combining Testing and Proving

The idea of combining testing and proving is not new, there are already testing frameworks for various proof assistants including Isabelle/HOL [4, 5] and Agda [8]. In a way, property-based testing resembles formal verification techniques, only that we are looking to disprove a property rather that prove it, and the tested properties are executable rather than declarative. In particular, RPBT can complement formal verification and reduce its overall cost. Property-based testing can be used as a counterexample finder to conjectures before commencing a time consuming proof. Detection of bugs in definitions and properties at the early stages of development can save a considerable amount of time and significantly reduce failed proof attempts. Coming up with the correct set of lemmas and the correct inductive invariants can be a hard and speculative procedure and discovering flaws through failed proof attempts comes at a very high cost, as each iteration is associated with the cost of trying to prove a wrong conjecture. Finding design flaws by testing can be much more cost-effective and can allow a much more rapid iteration on designs.

In this report, we are exploring another way of how testing and proving can be combined focusing on how formal verification can be used to provide formal guarantees about the quality of testing. More specifically our framework automatically maps generators to sets of values and executable property checkers to logical predicates. We can then form a *logical formula*, expressed as a Coq proposition, that precisely describes the property under test. We use the set of outcomes semantics to prove soundness and completeness for generators and the logical formula to prove that what our testing indeed corresponds to the desired high-level specification.

The intuitive guarantee we get from this method is that if we could enumerate the complete output space of our generators without producing any counterexamples then we would have a proof by exhaustion for the desired specification. While it is almost always impractical to completely enumerate the set of outcomes of generators, as it can very well be infinite, with this guarantee we can have confidence that we are indeed testing the right conjecture.

We built our framework on top of QuickChick, an existing RPBT plugin for Coq, which we will describe in the following section.

Chapter 3

The QuickChick Plugin for Coq

QuickChick is prototype RPBT plugin for Coq. Being a QuickCheck clone, it provides most of the functionality of QuickCheck. The largest part of QuickChick is implemented in Gallina, Coq's purely functional programming language, but the plugin relies on extraction to OCaml for acquiring and passing random seeds, for random generation of elements of primitive types, and for efficient execution and tracing of programs and property checkers.

As we discussed above, the main reason for having such a tool embedded in a proof assistant is testing the validity of conjectures. However, QuickChick currently does not facilitate any kind of testing for the non-executable part of Gallina and specifications have to be written in the executable part. This means that, in order to test the validity of lemmas that are expressed as declarative specifications, one needs to manually write and maintain equivalent executable variants for the lemmas under test. Given this restriction of the framework, the biggest advantage of QuickChick is that it eliminates the manual effort to extract executable properties in Haskell and run QuickCheck. Our extension to the framework provides us with one more argument of having such a tool embedded in a proof assistant.

Before we explain our verification framework in chapter 4, we first present the architecture of the QuickChick plugin, which closely follows the one of QuickCheck.

3.1 Generators

3.1.1 Representation

Generators are represented internally using the type `Gen`. `Gen` is a type family (a type parameterized by a another type). We define `Gen A` as an inductive type with only one constructor that wraps a function, which given a random seed and a size parameter returns an element of the type `A`.

```
Axiom RandomGen : Type.

Inductive Gen (A : Type) : Type :=
  | MkGen : (RandomGen → nat → A) → Gen A.
```

Listing 3.1: Generator representation

`RandomGen` is the type of the random seed. As QuickChick rely on extraction for seed generation, it is only stated as an axiom in Coq, omitting the actual definition. When extracting to OCaml `RandomGen` is mapped to `Random.State.t`.

The size parameter is often used by generators as an upper bound for the size of generated elements. Although there is no obligation whatsoever on the way a generator can use this size parameter, some may even ignore it, it is a good a practice to limit the size of the generated elements for efficiency reasons. Without the use of an upper bound some generators for recursive types may have some probability of never terminating. QuickChick will start by passing a small size parameter and gradually increase it up to a predefined bound.

3.1.2 Generators for Primitive Types

QuickChick relies on OCaml for generation of primitive data types, such as `Nat`, `Int` and `Bool`. In Coq, for each of these types we use an axiom to specify a function that given a range (l, h) and a seed it generates a random values from the closed interval $[l, h]$. More complex generators are built on top of these primitive ones. The type class `Random` with a method `randomR` provides a common interface for the functionality described above (listing 3.2). Since this function takes as an argument a pair that corresponds to a range we need to have an ordering relation for the types that are instances of this class. For this reason we make this type class a subclass of `OrdType` class, which provides a binary relation that satisfies reflexivity, transitivity and antisymmetry. This means that instances of the `Random` type class can be only types with a partial order. Note that calling `RandomR` with an invalid range, for which it holds that $h < l$, is an unspecified behavior that depends from the target language we are extracting to. Currently, since we are extracting to OCaml, such a call causes an invalid-argument exception.

3.1.3 Generators Interface

QuickChick provides a large library of generator combinators that can be instantiated to produce custom generators. Generator combinators fall into two categories: primitive and derived ones. Primitive combinators depend from the internal representation of generators, as they are using the `MkGen` constructor directly. On the other hand, derived combinators call the primitives ones but don't directly use the `MkGen` constructor. This distinction is important for the way we will later give dual semantics to these combinators.

The typed interface of the primitive combinators is shown in listing 3.3. Although `Gen` implements the monadic interface, monad laws are not directly provable. For instance, it is easy to see that a generator `g` and the generator `bindGen g returnGen` are not propositionally equal by unfolding the definitions or even extensionally equal by running the two generators. However the monad laws hold for equality of the underlying probability distributions.

It may seem counterintuitive that `fmapGen` is considered a primitive combinator, as it can be easily implemented using `bindGen` and `returnGen`. However, for efficiency reasons `fmapGen` is implemented in a primitive way without using other combinators. The resulting probability distribution is the same with the one we would have obtained with the standard implementation.

The derived combinators are listed in listing 3.4.

3.1.4 Arbitrary

Arbitrary is a QuickCheck type class (listing 3.5) that provides a common interface for types we can generate random elements. Arbitrary class has two fields `arbitrary`, which is the generator

```
Class OrdType (A: Type) :=
  {
    leq     : A → A → bool;
    refl    : ∀ a, leq a a;
    trans   : ∀ a b c, leq b a → leq a c → leq b c;
    antisym : ∀ a b, leq a b → leq b a → a = b
  }.

Instance OrdBool : OrdType bool := { ...  }.

Instance OrdNat : OrdType nat := { ...  }.

Instance OrdZ : OrdType Z := { ...  }.

Axiom randomRBool : bool * bool → RandomGen → bool * RandomGen.
Axiom randomRNat  : nat  * nat  → RandomGen → nat * RandomGen.
Axiom randomRInt  : Z    * Z    → RandomGen → Z  * RandomGen.

Class Random (A : Type) :=
  {
    randomR : A * A → RandomGen → A * RandomGen;
  }.

Instance Randombool : Random bool :=
  {
    randomR := randomRBool;
  }.

Instance Randomnat : Random nat :=
  {
    randomR := randomRNat;
  }.


Instance RandomZ : Random Z :=
  {
    randomR := randomRInt;
  }.
```

Listing 3.2: Random type class

```
(* Monadic return *)
returnGen
     : ∀ A : Type, A → Gen A

(* Monadic bind *)
bindGen
     : ∀ A B : Type, Gen A → (A → Gen B) → Gen B

(* Maps a function over a generator *)
fmapGen
     : ∀ A B : Type, (A → B) → Gen A → Gen B

(* Generates a random element in a given inclusive range. The type of
   the element must be an instance of the Random type class *)
choose
     : ∀ A : Type, Random A → A * A → Gen A

(*  A function used to construct generators that depend on the runtime
    size parameter *)
sized
     : ∀ A : Type, (nat → Gen A) → Gen A

(* Overrides the runtime size parameter with the given constant size.
   Only the first application has effect. *)
resize
     : ∀ A : Type, nat → Gen A → Gen A

(* Promotes a monadic generator to a generator of monadic values *)
promote
     : ∀ (M : Type → Type) (A : Type),
       ((Gen A → A) → M (Gen A) → M A) → M (Gen A) → Gen (M A)

(* Runs a generator and returns a list of example values. *)
sample
     : ∀ A : Type, Gen A → list A
```

Listing 3.3: Primitive generator combinators

```
(* Monadic lifting *)
liftGen
    : ∀ A B : Type, (A → B) → Gen A → Gen B

liftGen2
    : ∀ A1 A2 R : Type, (A1 → A2 → R) → Gen A1 → Gen A2 → Gen R

liftGen3
    : ∀ A1 A2 A3 R : Type,
      (A1 → A2 → A3 → R) → Gen A1 → Gen A2 → Gen A3 → Gen R

liftGen4
    : ∀ A1 A2 A3 A4 R : Type,
      (A1 → A2 → A3 → A4 → R) →
      Gen A1 → Gen A2 → Gen A3 → Gen A4 → Gen R

liftGen5
    : ∀ A1 A2 A3 A4 A5 R : Type,
      (A1 → A2 → A3 → A4 → A5 → R) →
      Gen A1 → Gen A2 → Gen A3 → Gen A4 → Gen A5 → Gen R

(* Monadic sequencing. Evaluates each generator from left to right and
   collects the results *)
sequenceGen
    : ∀ A : Type, list (Gen A) → Gen (list A)

foldGen
    : ∀ A B : Type, (A → B → Gen A) → list B → A → Gen A

(* Generates a value that satisfies a boolean predicate. Returns
   None if it fails to find one. *)
suchThatMaybe
    : ∀ A : Type, Gen A → (A → bool) → Gen (option A)

(* Randomly chooses one of the generators in the given list. Should
   the list be empty it returns the default one *)
oneof
    : ∀ A : Type, Gen A → list (Gen A) → Gen A

(* Chooses one of the given generators with a probability that
   corresponds to the given weights. Should the list be empty or all
   the weights set to zero it returns the default generator *)
frequency
    : ∀ A : Type, Gen A → list (nat * Gen A) → Gen A

(* Returns a list of random length using the given generator *)
listOf
    : ∀ A : Type, Gen A → Gen (list A)

(* Returns a generator that randomly chooses one of the elements of
   the given list of the default element if the list is empty *)
elements
    : ∀ A : Type, A → list A → Gen A
```

Listing 3.4: Derived combinators

for the given type and `shrink` which given an element returns a list of immediate shrinks.

```
Class Arbitrary (A : Type) : Type :=
{
  arbitrary : Gen A;
  shrink    : A → list A
}.
```

Listing 3.5: Arbitrary type class

3.2 Property Checkers

3.2.1 Property

Checkers are represented internally by the type `Property`. Results are represented by the type `Result`, a record keeping information about the outcome of the testing, the expected outcome. the reason of failing, etc.. In order for something to be testable it must to produce test results from randomly generated inputs, thus `Property` represents a probabilistic computation in the `Gen` monad that produces a `Result`. More precisely, `Property` is a generator of lazy rose[1] `Result` trees (listing 3.6). The reason of keeping a rose tree instead of just a `Result` is to keep track of the results of shrinking. The children of each node are the testing results for all the immediate shrinks that can produced from input data for of this node. Nevertheless, in order to inspect the result of testing is suffices to look only at the root as shrinking will only take place if this result is a failure.

```
Record Result :=
  MkResult {
      ok : option bool;
      expect : bool;
      reason : string;
      interrupted : bool;
      stamp : list (string * nat);
      callbacks : list Callback
    }.

Inductive Rose (A : Type) : Type :=
  MkRose : A → Lazy (list (Rose A)) → Rose A.

Record QProp : Type := MkProp
{
  unProp : Rose Result
}.

Definition Property := Gen QProp.
```

Listing 3.6: Property

[1]Rose trees or Multi-way trees are trees with arbitrary number of subtrees in each node

3.2.2 Testable

Testable is the class of things that can be checked, i.e. turned into a Checker. In order for a type to be an instance of the type class Testable it has to provide a function from an element of this type to a Property. This mechanism provides an automated way of deriving a Property. However, the user is free to construct their own checkers by using a library of combinators. In listing 3.7 we can see the type class and some significant instances.

In order for a function to be an instance of Testable then the types of its arguments have to be instances of the Arbitrary type class. If not such instance exists for this type then no instance for Testable can be derived. The generator and the shrinker provided by the Arbitrary interface are the default ones but one can override them by specifying others using forAll and forAllShrink combinators.

```
Definition Property := Gen QProp.
Class Testable (A : Type) : Type :=
  {
    property : A → Property
  }.

Instance testResult : Testable Result :=
  {|
    property r := returnGen (MkProp (returnRose r))
  |}.

Instance testBool : Testable bool :=
  {|
    property b := property (liftBool b)
  |}.


Instance testFun {A prop : Type} `{Show A}
         `{Arbitrary A} `{Testable prop} : Testable (A → prop) :=
  {|
    property f := forAll show arbitrary f
  |}.
```

Listing 3.7: Testable type class

3.3 Running Tests

Once generators and checkers are defined one can start testing. A number of run-time parameters are used to specify the maximum number of successful and discarded test cases, the maximum size bound, etc. Those parameters are gathered in a record Args (listing 3.10). One can use either the quickCheck function to run testing using the default parameters of quickCheckWith to in order to provide custom parameters (listing 3.11).

```
(* Lifts a boolean to a result. The boolean value is added to the ok
   field of the Result *)
liftBool : bool → Result

(* Maping functions *)

mapProp
     : ∀ P : Type, Testable P → (QProp → QProp) → P → Property

mapRoseResult
     : ∀ P : Type,
       Testable P → (Rose Result → Rose Result) → P → Property

mapTotalResult
     : ∀ prop : Type,
       Testable prop → (Result → Result) → prop → Property
```

Listing 3.8: Property lifting and mapping

```
(* adds a callback *)
callback
     : ∀ prop : Type, Testable prop → Callback → prop → Property

(* Given a Testable type prop, a shrinker for elements o type A, an
   element a of type A and a function from A to prop, shrinks the
   argument if (f a) fails. *)
shrinking
     : ∀ prop A : Type,
       Testable prop → (A → list A) → A → (A → prop) → Property

(* Prints a message if testing fails. *)
printTestCase
     : ∀ prop : Type, Testable prop → String.string → prop → Property

whenFail
     : ∀ prop : Type, Testable prop → (nat → nat) → prop → Property

(* Uses a test case generator to test a function A → prop *)
forAll
     : ∀ A prop : Type,
       Testable prop →
       (A → String.string) → Gen A → (A → prop) → Property

(* The same with forAll, but tries to shrink the argument of failing
   test cases. *)
forAllShrink
     : ∀ A prop : Type,
       Testable prop →
       (A → String.string) →
       Gen A → (A → list A) → (A → prop) → Property

(* Attaches a label to a property *)
label
     : ∀ prop : Type, Testable prop → String.string → prop → Property

(* Labels a property with a value.
   collect x = label (show x) *)
collect
     : ∀ A prop : Type,
       Show.Show A → Testable prop → A → prop → Property

(* Labels a property according to a boolean condition. *)
classify
     : ∀ prop : Type,
       Testable prop → bool → String.string → prop → Property

cover
     : ∀ prop : Type,
       Testable prop → bool → nat → String.string → prop → Property
```

Listing 3.9: Property combinators

```
Record Args := MkArgs {
  (* Specified if we want to repeat a previous test *)
  replay     : option (RandomGen * nat);
  (* Maximum number of test cases before success *)
  maxSuccess : nat;
  (* Maximum number of discard before giving up *)
  maxDiscard : nat;
  (* Maximum number of shrinks *)
  maxShrinks : nat;
  (* Maximum sizeparametr to be passed to a generator *)
  maxSize    : nat;
  (* Verbosity *)
  chatty     : bool
}.
```

Listing 3.10: Parameters for running the tests

```
(* Run test using the default parameters*)
quickCheck
     : ∀ prop : Type, Testable prop → prop → Result

(* Run test using custom parameters *)
quickCheckWithResult
     : ∀ prop : Type, Testable prop → Args → prop → Result
```

Listing 3.11: Functions for running the tests

Chapter 4

A Framework for Verified Testing in Coq

The first step towards mapping checkers to logical propositions is to able to reason about the values that can be produced by the generators used. To this end, we will abstract from the internal generator representation making generators parametric in the generator type constructor.

4.1 Generators

In order to be able to reason about the output space of the generators we map them to the sets of values that have non-zero probability of being generated. This is only an abstraction of the underlying probability distribution but it allows us to prove statements about the exact set of values that can be produced. Using the set of outcomes semantics of a generator we can prove that it is sound with respect to a specification, i.e. all the results produced by the generator satisfy the specification, and complete with respect to a specification, i.e. all the values that satisfy the specification can be generated.

Definition 4.1 (Soundness Definition)**.** *A generator is sound w.r.t. a predicate P, if its set of outcomes G satisfies the following proposition:*

$$\forall\, x,\ x \in G \rightarrow P(x)$$

Definition 4.2 (Completeness Definition)**.** *A generator is complete w.r.t. a predicate P, if its set of outcomes G satisfies the following proposition:*

$$\forall\, x,\ P(x) \rightarrow x \in G$$

We can combine definitions 4.1 and 4.2 to form a correctness statement about generators that completely specifies the set of outcomes. A generator is correct with respect to a predicate if it can generate all the values that satisfy the predicate and only them.

Definition 4.3 (Correctness Definition)**.** *A generator is correct w.r.t a predicate P if its set of outcomes G satisfies the following proposition:*

$$\forall\, x,\ P(x) \leftrightarrow x \in G$$

Example 4.4 (Generator of even numbers)**.** *Assume that we have a generator which we claim that produces all the possible even numbers and only them. We can formally verify our claim by proving the following proposition for its set of outcomes G_2:*

$$\forall\, x,\ (2 \mid x) \leftrightarrow (x \in G_2)$$

4.1.1 Set of Outcomes Monad

The representation of sets of outcomes we use in Coq has to support infinite sets (because the output space of generators can be infinite) and has to be targeted at proofs, not at efficient evaluation. We thus represent a set of elements of type `A` as a constructive characteristic function; i.e. function from an element from `A` to a Coq proposition (the Coq type `Prop`). In order to prove that an element x belongs to a set P we have to prove the proposition $P\ x$ obtained by applying P to x. With this representation we write the application $P\ x$ instead of the more standard $x \in P$ notation. This representation allows us to easily work with infinite and uncomputable sets. Although this representation may not seem very intuitive at first to people not familiar with Coq, it is quite natural as predicates are very often used to define sets in a very compact way with the set-builder notation. For instance, we can define the set of the natural even numbers as $\{x \mid (2 \mid x)\}$, which is isomorphic to the characteristic function $\lambda x.(2 \mid x)$.

We can now define set equality and set inclusion. We define equality using functional and propositional extensional equality on the predicates that represent the sets. Listing 4.1 shows the corresponding Coq code.

Definition 4.5 (Set equality)**.** *Two sets S_1, S_2 are equal, written $S_1 \longleftrightarrow S_2$, if*

$$\forall\, x,\ S_1\ x\ \leftrightarrow\ S_2\ x$$

Definition 4.6 (Set inclusion)**.** *A set S_1 is a subset of S_2 if for each value the proposition that results from the application S_1 that value implies the proposition that results from the application of S_2 to the same value:*

$$\forall\, x,\ S_1\ x\ \rightarrow\ S_2\ x$$

Example 4.7 (The set of even natural numbers)**.** *We can define the set of even numbers as a predicate:*

$$Even\ \equiv\ fun\,(x\ :\ nat) \Rightarrow\ 2 \mid x$$

We can now prove that 42, for instance, belongs to the set by proving the proposition

$$Even\ 42$$

which is by definition equal to

$$2 \mid 42$$

which trivially holds.

We give the monadic interface to the sets and we prove that the monad laws hold with respect to set equality. We define *return* as the function which given an element a returns the singleton set:

$$return\ a \equiv\ \{x \mid x = a\}$$

We define *bind* as the function which, given a set S and a function f, which maps each element of S to a set with elements from T, returns the set of all elements x, for which there exists an element $s \in S$ such that $x \in f\ s$.

$$bind\ S\ f \equiv\ \{x \mid \exists s,\ S\ s\ \wedge\ f\ s\ x\}$$

In other words, *bind* returns the union of the sets that we obtain by applying f to each element of S.

$$bind\ S\ f \longleftrightarrow \bigcup_{s \in S} f\ s$$

```
(* Set representation *)
Definition Pred (A : Type) := A → Prop.

(* Set equality *)
Definition set_eq {A} (m1 m2 : Pred A) :=
  ∀ A, m1 A ↔ m2 A.

Infix "⟷ " := set_eq (at level 70, no associativity) : pred_scope.

(* Set inclusion *)
Definition set_incl {A} (m1 m2 : Pred A) :=
  ∀ A, m1 A → m2 A.

(* Returns the singleton set *)
Definition returnP {A} (a : A) : Pred A :=
  fun x ⇒ eq a x.

Definition bindP {A B} (g : Pred A) (f : A → Pred B) : Pred B :=
  fun b ⇒ ∃ a, g a ∧ f a b.

Lemma left_identity :
  ∀ {A B} (f : A → Pred B) (a : A),
    (bindP (returnP a) f) ⟷ (f a).

Lemma right_identity :
  ∀ {A} (m: Pred A),
    (bindP m returnP) ⟷ m.

Lemma associativity :
  ∀ {A B C} (m : Pred A) (f : A → Pred B) (g : B → Pred C),
        (bindP (bindP m f) g) ⟷ (bindP m (fun x ⇒ bindP (f x) g)).
```

Listing 4.1: Set representation

4.1.2 Axiomatization

We want to be able to instantiate each generator with both executable and set of outcomes semantics. We do that by abstracting over the `Gen` monad, defined in listing 3.1. Generators will be parametrized by the type of monad which can be either instantiated with the concrete `Gen` monad or with `Pred`, the sets of outcomes monad. When instantiated with `Gen` a generator can be used for the actual generator and when instantiated with `Pred` can b mapped to its set of outcomes. From now on, we will attach the suffix `Gen` only to functions that abstract over the generator representation, the suffix `G` to functions that are concrete with regard to `Gen` type and the suffix `P` to functions that are concrete with regard to `Pred` type.

However, as we pointed out in section 3.1, there are a few combinators that are primitive in the sense that they are dependent from the generator representation and thus they cannot abstract over it. Those functions should have different implementations for each representation. In order to be to write combinators in a generic way that can be mapped to both executable and set of outcomes semantics both representations should have a common interface. In order to achieve this, we define a type class which specifies the functions each type should implement and we make both `Gen` and `Pred` instances of this type class. All other combinators can then be defined in terms of this set of primitive combinators. In the next paragraphs we will explain what functions should be included in the type class and how they are implemented for the set of outcomes representation.

bind and return

Both `bindGen` and `returnGen` are primitive combinators and should be defined in the type class. Listing 4.1 shows how these functions are implemented for the set of outcomes monad.

fmap

The combinator `fmapGen` is also a primitive one, as for type `Gen` it is not defined in terms of `bindGen` and `returnGen`, as we already discussed in section 3.1. For the `Pred` type constructor we define `fmapP` using `bindP` and `returnP`. We also prove the functor laws holds for set equality and up to functional extensional equality of the resulting functions. Listing 4.2 shows the implementation and the lemmas we proved.

```
Definition fmapP {A B} (f : A → B) (a : Pred A) : Pred B :=
  bindP a (fun a ⇒ returnP (f a)).

(* Functor laws *)
Lemma fmap_id:
  ∀ A (a: Pred A), (fmapP id a) ⟷ (id a).

Lemma fmap_composition:
  ∀ A B C (a : Pred A) (f : A → B) (g : B → C),
    (fmapP g (fmapP f a)) ⟷ (fmapP (fun x ⇒ g (f x)) a).
```

Listing 4.2: fmapP

choose

The combinator `choose` (listing 4.3) takes a pair of elements of a type `A`, that is instance of the type class `Random`, and returns a generator producing values that lie in the inclusive range that the given pair defines. Since type `A` needs to be an instance of the `Random` type class it also needs to be an instance of the `OrdType` type class. We can use the *less or equal than* relation that it is defined by the `OrdType` type class in order to express the set of outcomes of the resulting generator. A value belongs to the resulting set of outcomes if and only if it is less or equal than the first element of the pair and the second element of the pair is less or equal that the value. If $lo \leq hi$ doesn't hold then the resulting set is empty.

$$chooseP\ (lo, hi) \equiv \ \{x \ : \ lo \leq x \ \wedge \ x \leq hi\}$$

```
Definition chooseP {A : Type} `{Random A} (p : A * A) : Pred A :=
  fun a ⇒ leq (fst p) a ∧ leq a (snd p).
```

Listing 4.3: chooseP

sized

This combinator takes a generator that is parameterized by a natural number and returns the generator that depends internally from the global size parameter. Since the size parameter is arbitrary and every natural number could be used as a sized parameter it shouldn't restrict the set of possible outcomes. Intuitively, *sizedP f* is the union of the sets that result if we apply f to every natural number:

$$sizedP\ f \longleftrightarrow \bigcup_{n \in \mathbb{N}} f\ n$$

Formally, an element belongs to this set if and only of it exists a natural number n such that it belongs to $f\ n$:

$$sizedP\ f \equiv \ \{x \mid \exists n,\ \ f\ n\ x\}$$

```
Definition sizedP {A} (f : nat → Pred A) : Pred A :=
  fun a ⇒ ∃ n, f n a.
```

Listing 4.4: sizedP

resize

This combinator takes a generator and fixes the internal size parameter to a given value. This combinator is strongly connected to the `Gen` representation thus we will not add it to the common interface. This means that generators who use it will not be amenable to dual semantics. Nonetheless, this design choice does not restrict the expressiveness of the framework for writing generators. The need for using `resized` can be bypassed by fixing manually the parameters of generators that depend on numeric arguments.

sample

This function returns a list of example values of a generator and it is used for debugging purposes rather than building generators. We will not add this function to the common interface as we do not have a computational procedure that given a set can determine whether it is inhabited or not and if yes to return a subset of its elements.

suchThatMaybe

The function `suchThatMaybe` takes a generator of elements of type `A` and a boolean predicate on elements of type `A` and returns a generator for the type `option A`. If `Some a` is returned from this generator then `a` has to satisfy the boolean predicate. However, there is no guarantee whatsoever that if an element that satisfies the predicate exists it will be found, so if `None` is returned it does not imply that none of the elements satisfy the predicate. The set of outcomes of the returned generator includes `None`, that can be returned at any moment, and all the elements that are in the set of outcomes of the given generator and that additionally satisfy the predicate.

$$\mathit{suchThatMaybe}\ g\ P \equiv\ \{x \mid x = \mathit{None}\ \vee\ (\exists\, y,\ x = \mathit{Some}\ y\ \wedge\ g\ y\ \wedge\ P\ y)\}$$

```
Definition suchThatMaybeP {A} (g : Pred A) (f : A → bool) : Pred (option A) :=
  fun b ⇒ (b = None) ∨
           (∃ y, b = Some y ∧ g y ∧ f y).
```

Listing 4.5: suchThatMaybeP

The GenMonad Type Class

We can now define the type class `GenMonad` that provides a common interface for these primitive functions. We make both `Gen` and `Pred` instances of that type class. We can then define all the other combinators in terms of that common interface (section 4.1.3).

All the methods of the type class are implicitly quantified by the type constructor and the and the `GenMonad` instance of this type constructor. For example in listing 4.7 we can see the type of `bindGen`. The fist two arguments are implicit and can be automatically instantiated when a type constructor which is an instance of `GenMonad` is present in the context.

The set of outcomes denotation we give to each primitive combinator is an assumption we make about the semantics of each generator. We will not prove that the implementation for the combinators in terms on set of outcomes indeed corresponds to the implemetation in terms of concrete generators but we will use the instantiations of this type class as a trusted computing base in order to build the rest of the framework.

4.1.3 Derived Combinators

The remaining combinators can be built solely by using the functions of the `GenMonad` type class. Given that we do not need to change the implementation of any of the combinators the only thing we need to do is to make them parametric in the generator type constructor.

```
Class GenMonad M :=
  {
    bindGen : ∀ {A B : Type},  M A → (A → M B) → M B;
    returnGen : ∀ {A : Type}, A → M A;
    fmapGen : ∀ {A B : Type}, (A → B) → M A → M B;
    choose : ∀ {A} `{ Random A}, A * A → M A;
    sized : ∀ {A}, (nat → M A) → M A;
    suchThatMaybe : ∀ {A}, M A → (A → bool) → M (option A);
    promote : ∀  {A : Type}, (Rose (M A))  → M (Rose A)
  }.

Instance RealGen : GenMonad Gen :=
  {
    bindGen := @bindG;
    returnGen := @returnG;
    fmapGen := @fmapG;
    choose := @chooseG
    sized := @sizedG;
    suchThatMaybe := @suchThatMaybeG;
    promote := @promoteG
  }.

Instance PredMonad : GenMonad Pred :=
  {
    bindGen := @bindP;
    returnGen := @returnP;
    fmapGen := @fmapP;
    choose := @chooseP;
    sized := @sizedP;
    suchThatMaybe := @suchThatMaybeP;
    promote := @promoteP
  }.
```

Listing 4.6: The common interface

```
bindGen :
  ∀ M : Type → Type,
    GenMonad M → ∀ A B : Type, M A → (A → M B) → M B
```

Listing 4.7: The type of bindGen

We achieve this by placing the definitions of the combinators in a `Section` in which we assume to have in context an one-argument type constructor which is also an instance of the `GenMonad` type class. This technique can be seen in the listing 4.8. We write the two parameters in curly braces to denote that they are implicit. The primitive combinators, i.e. the methods of the type class, are automatically instantiated with the type constructor and the instance we assume in the context. When we close the section all the combinators written in it become implicitly parameterized by the type constructor and the `GenMonad` instance.

```
Section Utilities.
  Context {Gen : Type → Type}
          {H : GenMonad Gen}.


  Definition vectorOf {A : Type} (k : nat) (g : Gen A) : Gen (list A) :=
    fold_right
      (fun m m' ⇒
         bindGen m (fun x ⇒
         bindGen m' (fun xs ⇒ returnGen (cons x xs)))
      ) (returnGen nil) (nseq k g).


  ...
  ...
  ...

End Utilities.
```

Listing 4.8: Derived combinators

4.1.4 Lemma Library

Although at this point each generator is mappable to its set of outcomes in order to prove various statements about custom generators we will have to manually unfold the definitions of generator combinators which results in a very low level description of the set of outcomes. We provide a library of lemmas which provide a correctness proof for each generator combinator with respect to a higher level description of the set of outcomes of the resulting generator. These lemmas can be applied in a compositional way in order to prove correctness for other more complex generators.

Equality Lemmas

Fist, we provide a set of equality lemmas for primitive combinators that we can use for replacing primitive combinators with their definitions without having to unfold them, which requires to first unfold the type class method and then the type class instance. We do the same for lifting combinators to avoid unfolding them and then manually apply the lemmas for the primitive combinators in their definitions. The proofs of these lemmas are trivial and they only require unfolding the definitions.

```
Lemma bindGen_def :
  ∀ {A B} (g : Pred A) (f : A → Pred B),
    (bindGen g f) = fun b ⇒ ∃ a, g a ∧ f a b.

Lemma returnGen_def :
  ∀ {A} (a : A),
    returnGen a = fun x ⇒ a = x.

Lemma fmapGen_def :
  ∀ {A B} (f : A → B) (g : Pred A),
    fmapGen f g = fun b ⇒ ∃ a, g a ∧ f a = b.

Lemma choose_def :
  ∀ {A} `{Random A} (p : A * A),
    @choose Pred _ _ _ p = fun (a : A) ⇒ Random.leq (fst p) a ∧
                                         Random.leq a (snd p).

Lemma sized_def :
  ∀ {A} (g : nat → Pred A),
    sized g = fun a ⇒ ∃ n, g n a.

Lemma suchThatMaybe_def :
  ∀ {A} (g : Pred A) (f : A → bool),
    suchThatMaybe g f =
    fun b ⇒ (b = None) ∨
             (∃ y, b = Some y ∧ g y ∧ f y).
```

Listing 4.9: Equality lemmas for primitive combinators

```
Lemma liftGen_def :
  ∀ {A B} (f: A → B) (g: Pred A),
    liftGen f g =
    fun b ⇒
      ∃ a, g a ∧ f a = b.

Lemma liftGen2_def :
  ∀ {A B C} (f: A → B → C) (g1: Pred A) (g2: Pred B),
    liftGen2 f g1 g2 =
    fun b ⇒
      ∃ a1, g1 a1 ∧
                 (∃ a2, g2 a2 ∧ f a1 a2 = b).
Lemma liftGen3_def :
  ∀ {A B C D} (f: A → B → C → D)
          (g1: Pred A) (g2: Pred B) (g3: Pred C),
    liftGen3 f g1 g2 g3 =
    fun b ⇒
      ∃ a1, g1 a1 ∧
                 (∃ a2, g2 a2 ∧
                             (∃ a3, g3 a3 ∧ (f a1 a2 a3) = b)).

Lemma liftGen4_def :
  ∀ {A B C D E} (f: A → B → C → D → E)
          (g1: Pred A) (g2: Pred B) (g3: Pred C) (g4: Pred D),
    liftGen4 f g1 g2 g3 g4 =
    fun b ⇒
      ∃ a1, g1 a1 ∧
                 (∃ a2, g2 a2 ∧
                             (∃ a3, g3 a3 ∧
                                         (∃ a4, g4 a4 ∧
                                                     (f a1 a2 a3 a4) = b))).
Lemma liftGen5_def :
  ∀ {A B C D E G} (f: A → B → C → D → E → G)
          (g1: Pred A) (g2: Pred B) (g3: Pred C) (g4: Pred D) (g5: Pred E),
    liftGen5 f g1 g2 g3 g4 g5 =
    fun b ⇒
      ∃ a1,
        g1 a1 ∧
        (∃ a2, g2 a2 ∧
                  (∃ a3, g3 a3 ∧
                              (∃ a4, g4 a4 ∧
                                          (∃ a5, g5 a5 ∧
                                                      (f a1 a2 a3 a4 a5) = b)))).
```

Listing 4.10: Equality lemmas for lifting combinators

Set Equality Lemmas

When we instantiate combinators with the `Pred` constructor we can obtain set of outcomes semantics for the generators they return, the predicates that describe these sets of outcomes are fairly complex and difficult to understand and they often depend other combinators which have to be replaced manually with their definitions. We provide a set of lemmas that prove that the resulting set for each combinator is equal to a set that is described by high-level predicate that does not depend on other combinators, is easier to understand intuitively and more useful in proofs. Using this library of lemmas a proof for a complex generator can be built in a more compositional and incremental way avoiding proof duplication. Also, the proofs are more robust as they do not depend on the implementation of the generator combinators which can be subject to change over the time. In the next paragraphs we will explain further the lemmas we proved about each generator.

sequenceGen

Given a list of generators `gs` this combinator returns a generator of lists each element of which is generated by the generator in the corresponding position in the initial list. In order for a list to belong at the resulting set of outcomes must have length equal to the one of the initial list and each element of it should belong to the set of outcomes of the corresponding element of the given list. We use the function zip to pair each element of the generated list to the corresponding element of the initial list and state that the former should belong to the set of outcomes of the later.

```
Lemma sequenceGen_equiv :
  ∀ {A} (gs : list (Pred A)),
    sequenceGen gs ⟷ fun l ⇒ length l = length gs ∧
                                      ∀ x, In x (zip l gs) → (snd x) (fst x).
```

Listing 4.11: `sequenceGen` lemma

vectorOf

The combinator `VectorOf` takes a natural number `n` and a generator `g` and returns a generator that produces lists of length `n` whose elements are generated by the given generator. A list belongs to the set of outcomes of the returned generator if and only if has length `n` and its elements belong to the set of outcomes of `g`.

```
Lemma vectorOf_equiv:
  ∀ {A : Type} (k : nat) (g : Pred A),
    vectorOf k g ⟷ fun l ⇒ (length l = k ∧ ∀ x, In x l → g x).
```

Listing 4.12: `vectorOf` lemma

listOf

Given a generator `g` this combinator returns generator
g that produces lists of random length whose elements re generated by `g`. A list belongs to the

set of outcomes of the returned generator if and only if its elements belong to the set of outcomes of g.

```
Lemma listOf_equiv:
  ∀ {A : Type} (g : Pred A),
    listOf g ⟷ fun l ⇒ (∀ x, In x l → g x).
```

Listing 4.13: `listOf` lemma

oneOf

Given a list generators gs ad a default generator g this combinator returns a randomly chosen generator from the list or, should the list is empty, g. An element belongs to the sets of outcomes of the returned generator if and only if it belongs to the set of outcomes of a generator in the given list, or in the set of outcomes of g if the list is empty.

```
Lemma oneOf_equiv:
  ∀ {A} (l : list (Pred A)) (def : Pred A),
    (oneOf def l) ⟷
     (fun e ⇒ (∃ x, (In x l ∧ x e)) ∨ (l = nil ∧ def e)).
```

Listing 4.14: `oneOf` lemma

elements

This combinator takes a list of elements and a default element and returns a generator that randomly picks elements from the list or returns the default element if the list is empty. An element belongs to the sets of outcomes of the returned generator if and only if it is an element of the given list, or it is equal to the default element if the list is empty.

```
Lemma elements_equiv :
  ∀ {A} (l: list A) (def : A),
    (elements def l) ⟷ (fun e ⇒ In e l ∨ (l = nil ∧ e = def)).
```

Listing 4.15: `elements` lemma

frequency

This combinator takes a list l of pairs consisting of a natural number (frequency) and an element of type A and a default element a and returns a generator which picks an element of type A from the given list or the default element should the list is empty or all the frequencies are equal to zero. If $[(f_1, a_1), \ldots, (f_i, a_i), \ldots, (f_n, a_n)]$ is the given list then probability of an arbitrary element a_i to be generated is:

$$p_i = \frac{f_i}{\sum_{j=0}^{n} f_j}$$

An element a belongs to the set of outcomes of the generator if and only if exists $f \neq 0$ such that $(f, a) \in$ `l` or it is equal to the default element if the list is empty or the sum of all frequencies is 0.

```
Lemma frequency_equiv :
  ∀ {A} (l : list (nat * Pred A)) (def : Pred A),
    (frequency def l) ⟷
     fun e ⇒ (∃ (n : nat) (g : Pred A),
                 In (n, g) l ∧ g e ∧ n <> 0) ∨
              ((l = nil ∨ ∀ x, In x l → fst x = 0) ∧ def e).
```

Listing 4.16: `frequency` lemma

foldGen

This generator takes a function `f` that takes two arguments of type `A` and `B` and returns a generator of type `A`, a list of elements of type `B` and an element $\mathtt{a}_0$ of type `A`. It returns the generator $\mathtt{g}_n$, where n is the length of the given list, which is obtained as following:

$$\mathtt{g}_i = \mathtt{f}\ \mathtt{a}_{i-1}\ \mathtt{b}_i$$

where $\mathtt{b}_i$ is an element of the given list at position i and $\mathtt{a}_{i-1}$ is an element generated by $\mathtt{g}_{i-1}$ or the initial one if $i = 1$. In terms of sets of outcomes we can express the resulting sets with the following recursive formula:

$$\begin{aligned} g_i &= \{x \ : \ \exists\, a, g_{i-1}\ a \ \wedge \ f\ a\ b_i\ x\} && \text{when } i \neq 1 \\ g_1 &= \{x \ : \ f\ a_0\ b_1\ x\} \end{aligned}$$

We can use both `foldl` and `foldr` to write the above set in a compact way. The correspondence with the above definition is more obvious when we express the set with the use of `foldl`.

```
Lemma foldGen_left_equiv :
  ∀ {A B : Type} (f : A → B → Pred A) (bs : list B) (a0 : A),
    foldGen f bs a0 ⟷
    foldl (fun g b ⇒ fun x ⇒ ∃ a, g a ∧ f a b x) (eq a0) bs.

Lemma foldGen_right_equiv :
  ∀ {A B : Type} (f : A → B → Pred A) (bs : list B) (a0 : A),
    foldGen f bs a0 ⟷
    fun an ⇒
      foldr (fun b p ⇒ fun a_prev ⇒ ∃ a, f a_prev b a ∧ p a)
            (eq an) bs a0.
```

Listing 4.17: `foldGen` lemmas

4.1.5 A Motivating Example for Combinator Lemmas

We emphasize the value of this lemma library through an example. Let `genNat` be a generator of natural numbers and we want to build a generator for lists of natural numbers with length

5. We write those generators in a section assuming a type constructor which is instance of the `GenMonad` type class in context, as we can see in listing 4.18.

```
Section Generators.
  Context {Gen : Type → Type}
          {H : GenMonad Gen}.

  Definition genNat : Gen nat := sized (fun n ⇒ choose (0, n)).

  Definition genList5 : Gen (list nat) := vectorOf 5 genNat.

End Generators.
```

Listing 4.18: Generator for lists of length 5

The we can prove that `genList5` produces all the possible lists of natural numbers with length 5. First, we have prove that `GenNat` produces all the possible natural numbers by proving the lemma in listing 4.19. We can try to do this without using the equality lemmas for `choose` and `sized`.

```
Lemma genNat_correct:
  genNat ⟷ fun _ : nat ⇒ True.
```

Listing 4.19: Lemma for `genNat` correctness

Fist we try to unfold `genNat`, `sized` and `choose`. Then we have to unfold `PredMonad` instance and finally we have to unfold `chooseP` and `sizedP`. We can see the intermediate goals in listings 4.20 to 4.22. Instead of doing this we can simply replace the definitions of `sized` and `choose` using the corresponding equality lemmas and immediately reach the third goal. We can see the final proof in listing 4.23

```
1 subgoals, subgoal 1 (ID 5)

  ============================
   (let (_, _, _, _, sized, _, _) := PredMonad in sized) nat
     (fun n : nat ⇒
      (let (_, _, _, choose, _, _, _) := PredMonad in choose) nat
        Random.Randomnat (0, n)) ⟷ (fun _ : nat ⇒ True)
```

Listing 4.20: Goal after unfolding `genNat`, `sized` and `choose`

```
 1 subgoals, subgoal 1 (ID 6)

  ============================
   sizedP (fun n : nat ⇒ chooseP (0, n)) ⟷ (fun _ : nat ⇒ True)
```

Listing 4.21: Goal after unfolding `PredMonad`

We can now use this proof to prove that `genList5` generates all the possible list of natural number with length 5 and only them. The lemma is stated in listing 4.24. We can try to

```
1 subgoals, subgoal 1 (ID 7)

  ============================
   (fun a : nat ⇒
    ∃ n : nat,
      is_true (Random.leq (fst (0, n)) a) ∧
      is_true (Random.leq a (snd (0, n)))) ⟷ (fun _ : nat ⇒ True)
```

Listing 4.22: Goal after unfolding `chooseP` and `sizedP`

```
Lemma genNat_correct:
  genNat ⟷ fun _ : nat ⇒ True.
Proof.
  unfold genNat. rewrite sized_def.
  intros x. split; auto. intros _. ∃ x.
  rewrite choose_def. split; auto.
Qed.
```

Listing 4.23: Proof for `genNat` correctness

do the proof without using the set equality lemma for `vectorOf`. First, we unfold `genList5` and afterwards we unfold `vectorOf`. After some simplification we obtain the goal shown in listing 4.25. Proving such a goal would be a very tedious process as we would have to repeatedly apply the same lemmas for the used combinators. Instead we can use `vectorOf_equiv lemma` to obtain a very compact representation for the sets of outcomes and a much smaller and easily provable goal, which is shown in listing 4.26. The final proof is shown in listing 4.27.

```
Lemma genList5_correct:
  genList5 ⟷ fun l ⇒ length l = 5.
```

Listing 4.24: Lemma for `genList5` correctness

4.1.6 `Arbitrary` Type Class

The `Arbitrary` type class provides default generators for its instances. In order to be able to map these generators to set of outcomes we need to parameterize the `arbitrary` method with the generator type constructor. We change the definition of the type class as shown in listing 4.28. We instantiate the type class with generators that are parameterized by the generator type constructor.

We also provide lemmas for these generators proving that they can generate all the possible elements of the underlying type. In particular for `arbList`, that uses the `arbitrary` method to generate list elements, we prove that it can generate all the possible lists, if `arbitrary` for the type of the elements can generate all the possible values of the type. We can see to proofs in listing 4.30.

```
1 subgoals, subgoal 1 (ID 37)

  l : list nat
  ============================
   bindGen genNat
     (fun x : nat ⇒
      bindGen
        (bindGen genNat
           (fun x0 : nat ⇒
            bindGen
              (bindGen genNat
                 (fun x1 : nat ⇒
                  bindGen
                    (bindGen genNat
                       (fun x2 : nat ⇒
                        bindGen
                          (bindGen genNat
                             (fun x3 : nat ⇒
                              bindGen (returnGen nil)
                                (fun xs : list nat ⇒ returnGen (x3 :: xs))))
                          (fun xs : list nat ⇒ returnGen (x2 :: xs))))
                    (fun xs : list nat ⇒ returnGen (x1 :: xs))))
              (fun xs : list nat ⇒ returnGen (x0 :: xs))))
        (fun xs : list nat ⇒ returnGen (x :: xs)))  l ↔
   length l = 5
```

Listing 4.25: Goal after unfolding `vectorOf`

```
  l : list nat
  ============================
   length l = 5 ∧ (∀ x : nat, In x l → genNat x) ↔ length l = 5
```

Listing 4.26: Goal after using the set equality lemma

```
Lemma genList5_correct:
  genList5 ⟷ fun l ⇒ length l = 5.
Proof.
  intros l. unfold genList5. rewrite (vectorOf_equiv _ _ _). split.
  – intros [H _]; auto.
  – intros H. split; auto. intros x _. by apply genNat_correct.
Qed.
```

Listing 4.27: Proof for `genList5` correctness

```
Class Arbitrary (A : Type) : Type :=
  {
    arbitrary : ∀ {Gen : Type → Type} {H : GenMonad Gen}, Gen A;
    shrink    : A → list A
  }.
```

Listing 4.28: New definition of the `Arbitrary` type class

```
Section ArbitrarySection.
  Context {Gen : Type → Type}
          {H : GenMonad Gen}.

  Definition arbitraryBool := choose (false, true).

  Definition arbitraryNat :=
    sized (fun x ⇒ choose (0, x)).

  Definition arbitraryZ :=
    sized (fun x ⇒
             let z := Z.of_nat x in
             choose (−z, z)%Z).

  Definition arbitraryList {A : Type} {Arb : Arbitrary A} :=
    listOf arbitrary.

End ArbitrarySection.

Instance arbBool : Arbitrary bool :=
  {|
    arbitrary := @arbitraryBool;
    shrink  := shrinkBool
  |}.

Instance arbNat : Arbitrary nat :=
  {|
    arbitrary := @arbitraryNat;
    shrink x := shrinkNat x
  |}.

Instance arbInt : Arbitrary Z :=
  {|
    arbitrary := @arbitraryZ;
    shrink := shrinkZ
  |}.

Instance arbList {A : Type} {Arb : Arbitrary A} : Arbitrary (list A) :=
  {|
    arbitrary g H := @arbitraryList g H A Arb;
    shrink := shrinkList shrink
  |}.
```

Listing 4.29: Instance definitions for the `Arbitrary` type class

```
Lemma arbBool_correct:
  arbitrary ⟷ (fun (_ : bool) ⇒ True).

Lemma arbNat_correct:
  arbitrary ⟷ (fun (_ : nat) ⇒ True).

Lemma arbInt_correct:
  arbitrary ⟷ (fun (_ : Z) ⇒ True).

Lemma arbList_correct:
  ∀ {A} {H : Arbitrary A},
    (arbitrary ⟷ (fun (_ : A) ⇒ True)) →
    (arbitrary ⟷ (fun (_ : list A) ⇒ True)).
```

Listing 4.30: Proofs for `arbitrary` generators

4.2 Checkers

Checkers are represented internally with the type `Property`. In order for something to be testable it should be able to be turned into a `Property`. The type class testable provides some automation for turning testable things into a `Property` by declaring through type class instances a canonical way to turn something testable into a `Property`. In order to map checkers to propositions we should be able to turn a `Property` into a proposition.

4.2.1 Semantics

`Property` represents is a non-deterministic computation of elements of type `QProp`, i.e. it is a generator of elements of `QProp`. The type `QProp` wraps a record around rose trees of results, as we have already seen in listing 3.6. We can learn the result of testing just by looking at the root of the tree, since the other levels of the tree are only used to facilitate shrinking, which doesn't change the testing outcome and only happens if a failure is encountered. Intuitively, we can map a `Property` to a proposition by requesting all the results that belong to the set of testing outcomes to be successful.

Since `Property` is a type synonym for `Gen QProp` and in order to map it to propositions we need to abstract over the generator by making `Property` parametric on the generator type constructor. We change the definition of `Property` as shown in listing listing 4.31. As in generators, we put `Property` combinators in a section in which we assume in context a type constructor which is instance of the `GenMonad` type class.

```
Definition Property := Property Pred.
```

Listing 4.31: Definition of `Property`

We should also define what means for the testing outcome to be successful. A `Result` is correct either when the field `ok` is `None` which denotes a discarded test case, or then the field `ok` is equal to `Some b` and the boolean value `b` is equal to the value of the `expect` field, which denotes the expected testing result. When `b` does not coincide with `expect` then we have a failed `Result`. The testing outcome, which is of type `QProp` is successful when the root node of the rose tree is a successful. The two definitions can be seen in listing 4.32.

```
Definition resultSuccessful (r : Result) : bool :=
  match r with
    | MkResult (Some res) expected _ _ _ _ ⇒
      res == expected
    | _ ⇒ true
  end.
```

Listing 4.32: Successful `Result` definition

Using these definitions, we can map a `Property` to a proposition by stating that for all the values that belong to its set of outcomes are successful. The function that maps a `Property` to a proposition is named `semProperty` and we can see its definition in listing 4.33. We can now give semantics to each element of a type that is instance to the `Testable` type class by turning it into a `Property` and calling the `semProperty` function.

```
Definition success qp :=
  match qp with
    | MkProp (MkRose res _) ⇒ resultSuccessful res
  end.

Definition semProperty (P : Property) : Prop :=
  ∀ qp, P qp → success qp = true.

Definition semTestable {A : Type} {_ : Testable  A} (a : A) : Prop :=
  semProperty (property a).
```

Listing 4.33: Semantics for `Property`

The proposition we obtain by this method completely describes the conjecture under test and by proving it equivalent to a high-level specification we are proving that the conjecture under test indeed corresponds to the intended high-level specification.

4.2.2 Lemma Library

Although the type class `Testable` provides an automated way of deriving `Properties`, the later can be also formed by using a library of combinators. We provide a set of lemmas that prove that the `Property` obtained from each combinator is equivalent to a less complex one. We can use this set of lemmas to prove statements about the resulting properties without having to unfold the definitions of combinators.

Identity Lemmas

A number of `Property` combinators are intended for instrumentation purposes and thus they do not affect the testing outcome. We provide a set of lemmas that state that the proposition we obtain from the `Property` before applying these combinators is equivalent to the one we obtain from the `Property` that is returned after the application of the combinator.

```
Lemma semCallback_id:
  ∀ {prop : Type} {H : Testable prop} (cb : Callback) (p : prop),
    semProperty (callback cb p) ↔ semTestable p.

Lemma semWhenFail_id:
  ∀ {prop : Type} {H : Testable prop} (s : String.string) (p : prop),
    semProperty (whenFail s p) ↔ semTestable p.

Lemma semPrintTestCase_id:
  ∀ {prop: Type} {H : Testable prop} (s: String.string) (p: prop),
    semProperty (printTestCase s p) ↔ semTestable p.

Lemma semShrinking_id:
  ∀ {prop A : Type} {H : Testable prop}
         (shrinker : A → list A) (x0 : A) (pf : A → prop),
    semProperty (shrinking shrinker x0 pf) ↔ semTestable (pf x0).

Lemma semCover_id:
  ∀ {prop : Type} {H : Testable prop} (b: bool) (n: nat)
         (s: String.string) (p : prop),
    semProperty (cover b n s p) ↔ semTestable p.

Lemma semClassify_id:
   ∀ {prop : Type} {H : Testable prop} (b: bool) (s: String.string)
          (p : prop),
    semProperty (classify b s p) ↔ semTestable p.

Lemma semLabel_id:
   ∀ {prop : Type} {H : Testable prop} (s: String.string)
          (p : prop),
    semProperty (label s p) ↔ semTestable p.

Lemma semCollect_id:
   ∀ {prop : Type} {H : Testable prop} (s: String.string)
          (p : prop),
    semProperty (collect s p) ↔ semTestable p.

Lemma mapTotalResult_id:
  ∀ {prop : Type} {H : Testable prop} (f : Result → Result) (p : prop),
    (∀ res, resultSuccessful res = resultSuccessful (f res)) →
    (semProperty (mapTotalResult f p) ↔ semTestable p).
```

Listing 4.34: Identity lemmas for `Property` combinators

`forAll`

This combinators take a generator of elements of type `A` and a function `f` from `A` to a type that is instance of the testable class and return a `Property` that tests the given function using the given generator. It must also be provided with a function for printing elements of type `A`, in order to print test cases. We prove that the proposition that we obtain from the returned `Property` is equivalent to the proposition that states that all elements `a` that can be generated from the generator, satisfy the proposition that we obtain from `f a`.

```
Lemma semForAll :
  ∀ {A prop : Type} {H : Testable prop}
         show (gen : Pred A) (f : A → prop),
    semProperty (forAll show gen f) ↔
    ∀ a : A, gen a → semTestable (f a).

Lemma semForAllShrink:
  ∀ {A prop : Type} {H : Testable prop}
         show (gen : Pred A) (f : A → prop) shrinker,
    semProperty (forAllShrink show gen shrinker f) ↔
    ∀ a : A, gen a → semTestable (f a).
```

Listing 4.35: Lemmas for `forAll` and `forAllShrink`

`implication`

The combinator `implication` takes a boolean parameter and a parameter of a `Testable` type and returns a `Property` that discards all the test case if the boolean parameter is not `true`. We prove that the resulting proposition is equivalent to `True` if the boolean parameter is `false` otherwise is equivalent to the proposition that we can obtain from the given `Testable` element.

```
Lemma semImplication:
      ∀ {prop : Type} {H : Testable prop}
             (p : prop) (b : bool),
        semProperty (b ==> p) ↔ b = true → semTestable p.
```

Listing 4.36: Lemma for `implication`

Lemmas for specific types

We also prove that the propositions we obtain by `semTestable` for some `Testable` instances are equivalent to some more intuitive and high level propositions. For example we prove that for boolean values the equivalent proposition is the proposition we get by setting the boolean value to be equal to `true` and for `Result` values the equivalent proposition is setting the result of the function `resultSuccessful` applied in the value to be equal to `true`. For functions from a type `A` which is instance of `Arbitrary` type class to a type `prop` which is instance of `Testable`, we prove that the proposition is equivalent to the one that states that all elements `a` that can be generated from the `arbitrary` method satisfy the proposition that we obtain from `f a`. For polymorphic functions we prove that the proposition we obtain is equivalent to the proposition we obtain if we instantiate the polymorphic type to `nat`. This is the strongest property we can obtain since by default polymorphic functions will only be tested for values of type `nat`.

```
Lemma semBool:
  ∀ (b : bool), semTestable b ↔ b = true.

Lemma semResult:
  ∀ (res: Result), semTestable res ↔ resultSuccessful res = true.

Lemma semUnit:
  ∀ (t: unit), semTestable t ↔ True.

Lemma semQProp:
  ∀ (qp: QProp), semTestable qp ↔ success qp = true.

Lemma semGen:
  ∀ (P : Type) {H : Testable P} (gen: Pred P),
    (semTestable gen) ↔ (∀ p, gen p → semTestable p).

Lemma semFun:
  ∀ {A prop : Type} {H1 : Show A} {H2 : Arbitrary A} {H3 : Testable prop}
         (f : A → prop),
    semTestable f ↔
    (∀ (a : A), (arbitrary : Pred A) a → semTestable (f a)).

Lemma semPolyFun:
  ∀ {prop : Type → Type} {H : Testable (prop nat)} (f : ∀ T, prop T),
    (semTestable f) ↔ (semTestable (f nat)).

Lemma semPolyFunSet:
  ∀ {prop : Set → Type} {H : Testable (prop nat)} (f : ∀ T, prop T),
    (semTestable f) ↔ (semTestable (f nat)).
```

Listing 4.37: Lemma for `implication`

4.2.3 `Provable` Type Class

We define the type class `Provable` to provide some automation for the above lemmas application. More specifically, `Provable` provides a proposition for all the types which are also instances of `Testable` and a proof that the aforementioned proposition is equivalent to the one we get by `semTestable` but more high-level and understandable. This proposition can also be obtained manually but it would require the application of the lemmas we described above. Since this type class require its instances to also be instances of the `Testable` type class we make it a subclass of it. The definition of the type class and its instances can be seen in listings 4.38 to 4.40.

```
Class Provable (A : Type) {H: Testable A} : Type :=
 {
    proposition : A → Prop;
    _ : ∀ a, proposition a ↔ semTestable a
  }.
```

Listing 4.38: `Provable` type class

```
Program Instance proveResult : Provable Result :=
  {|
    proposition := resultSuccessful
  |}.
Next Obligation.
  by rewrite semResult.
Qed.

Program Instance proveUnit : Provable unit :=
  {|
    proposition := fun _ => True
  |}.
Next Obligation.
  by rewrite semUnit.
Qed.

Program Instance proveQProp : Provable QProp :=
  {|
    proposition qp := success qp = true
  |}.
Next Obligation.
  by rewrite semQProp.
Qed.

Program Instance proveBool : Provable bool :=
  {|
    proposition b :=  b = true
  |}.
Next Obligation.
  by rewrite semBool.
Qed.
```

Listing 4.39: `Provable` instances

```
Program Instance proveGenProp {prop : Type} `{ Provable prop} :
  Provable (Pred prop) :=
  {|
    proposition g := (∀ p, g p → proposition p)
  |}.
Next Obligation.
  destruct H0 as [semP proof]. rewrite /proposition. split.
  - move ⇒ H'. apply semGen⇒ p Hgen. apply proof. by auto.
  - move ⇒ /semGen H' p Hgen. apply proof. by auto.
Qed.

Program Instance proveFun {A prop: Type} `{ Arbitrary A} `{ Show A}
        `{ Provable prop}: Provable (A → prop) :=
  {|
    proposition p :=
      (∀ a,
         @arbitrary _ _ Pred _ a →
         proposition (p a))
  |}.
Next Obligation.
  destruct H2 as [semP proof]. rewrite /proposition. split.
  - move⇒ H'. apply semFun ⇒ a' /H' Hgen.
    by apply proof.
  - move⇒ H' a' Hgen. apply proof. by apply semFun.
Qed.

Program Instance provePolyFun {prop : Type → Type} `{Provable (prop nat)} :
  Provable (∀ T, prop T) :=
  {
    proposition f := proposition (f nat)
  }.
Next Obligation.
  destruct H0 as [semP proof]. rewrite /proposition. split.
  - move⇒ /proof H'. by apply semPolyFun.
  - move⇒ /semPolyFun H'. by apply proof.
Qed.

Program Instance provePolyFunSet {prop : Set → Type} `{Provable (prop nat)} :
  Provable (∀ T, prop T) :=
  {
    proposition f := proposition (f nat)
  }.
Next Obligation.
  destruct H0 as [semP proof]. rewrite /proposition. split.
  - move⇒ /proof H'. by apply semPolyFunSet.
  - move⇒ /semPolyFunSet H'. by apply proof.
Qed.
```

Listing 4.40: `Provable` instances

4.2.4 A Motivating Example for Combinator Lemmas

We will emphasize the need for the lemma library and the `Provable` type class with an example. We write a checker that checks the property that for every list if we reverse it twice we will get the initial list. We express it as a boolean predicate that takes a list of natural numbers and checks if the property holds by using the SSReflectperator `==` for boolean equality. This can be automatically mapped into a `Property` as it is an instance of the `Testable` type class. For the list of natural numbers generation, we will use the default generators provided by the `Arbitraty` type class. In particular, `arbList` will be used for list which will use `arbNat` for number generators. We have already proved that `arbList` can generate all the possible list if the generator used for the elements if them can also generate all the possible elements, which is the case for `arbNat`. We write the checker in a section to parameterize it by the generator type constructor.

```
Section Properties.
  Context {Gen : Type → Type}
          {H : GenMonad Gen}.

  Definition prop_reverse := (fun (l : list nat) ⇒ rev (rev l) == l).
```

Listing 4.41: Checker for `rev`

Then, we can prove that a proposition corresponds to the conjecture under test by proving the lemma shown in listing 4.42.

```
Lemma prop_reverse_correct:
  (semTestable prop_reverse) ↔ (∀ (l : list nat), (rev (rev l) = l)).
```

Listing 4.42: Lemma for `rev` checker

In order to prove such a lemma we could start by unfolding `semTestable` and `semPropety` definitions. After that we would need to unfold `property` and thus the corresponding `Testable` instance. That would lead us to the goal shown in listing 4.43. Trying to prove that goal requires a lot of effort as we would have to reason about `forAll`. Even simplification at that point would lead to a very complex and long goal. Instead we can use `semFun` lemma to get rid of the outermost call to `semTestable` leading us to the goal we can see in listing 4.44. As the proof proceeds can use `semBool` eradicate the inner `semTestable` occurrence.

```
1 subgoals, subgoal 1 (ID 91)

  ============================
   (∀ qp : QProp,
    forAll show arbitrary shrink prop_reverse qp → success qp = true) ↔
   (∀ l : seq nat, rev (rev l) = l)
```

Listing 4.43: Goal after unfolding `semTestable`, `semPropety` and `property`

However, instead of manually applying the lemmas we can use the `proposition` method to get the corresponding proposition. The lemma to be proved should change to the one depicted in

```
1 subgoals, subgoal 1 (ID 103)

  ============================
   (∀ a : seq nat, arbitrary a → semTestable (prop_reverse a)) ↔
   (∀ l : seq nat, rev (rev l) = l)
```

Listing 4.44: Goal after using `semFun`

listing 4.45. Then after simplification we can obtain the goal shown in listing 4.46. The complete proof can be seen in listing 4.47.

```
Lemma prop_reverse_correct':
  (proposition prop_reverse) ↔ (∀ (l : list nat), (rev (rev l) = l)).
```

Listing 4.45: Lemma for `rev` checker using `Provable` type class

```
1 subgoals, subgoal 1 (ID 93)

  ============================
   (∀ a : seq nat, arbitraryList a → prop_reverse a = true) ↔
   (∀ l : seq nat, rev (rev l) = l)
```

Listing 4.46: Goal obtained by using `proposition`

```
Lemma prop_reverse_correct':
  (proposition prop_reverse) ↔ (∀ (l : list nat), (rev (rev l) = l)).
Proof.
  simpl. unfold prop_reverse. split.
  - intros H l. apply/eqP. apply H. apply arbList_correct; auto.
    split; auto. intros _. by apply arbNat_correct.
  - intros H l _. by apply/eqP.
Qed.
```

Listing 4.47: Lemma for `rev` checker using `Provable` type class

Chapter 5

Case Studies

At this chapter we will use our verification framework to prove correct a non-trivial generator for red-black trees and finally prove the executable checker we will write to test the implementation or red-black trees corresponds to a high level proposition. Furthermore, we will describe our experience from using our framework to verify the testing infrastructure of a more complex development of an information-flow control (IFC) machine.

5.1 Red-black Trees

A Red-black tree is a binary search tree, each node of which is additionally labeled with a color, namely red or black. It is a self-balancing data structure and the balance is preserved by enforcing the following invariants:

- The root is always black
- The leaves are empty and black
- For all the node the path to each possible leaf has the same number of black nodes
- Red nodes have only black children

From these invariants it follows that the longest path to a leave is at most twice as long as the shortest and this allows all operations to run in $\mathcal{O}(\log n)$ time, where n is the total number of nodes. An implementation in a purely functional setting has been proposed by Okasaki in [17] and verified in Coq by Appel in [2]. We will the implementation proposed in [17] the to write generators and checkers that will test that the red-black invariant is preserved by the `insertion` operation. That claim has already been verified in [2] however our intention is to demonstrate how testing infrastructure can be formally verified rather that find counterexamples in the implementation of red-black trees.

5.1.1 Representation

The representation of red-black trees is straightforward and only requires a small modification to the common binary tree data type in order to keep a color label in each node. For simplicity, we choose the key of the node to be an natural number.

```
Inductive color := Red | Black.

Inductive tree :=
  | Leaf : tree
  | Node : color → tree → nat → tree → tree.
```

Listing 5.1: Red-black trees data type

5.1.2 Declarative Definitions

As we can see in [2], the red-black tree invariant can be formalized as an inductive definition. We can see the invariant definition in listing 5.2.

```
Inductive is_redblack : tree → color → nat → Prop :=
  | IsRB_leaf: ∀ c, is_redblack Leaf c 0
  | IsRB_r: ∀ n tl tr h,
             is_redblack tl Red h → is_redblack tr Red h →
             is_redblack (Node Red tl n tr) Black h
  | IsRB_b: ∀ c n tl tr h,
             is_redblack tl Black h → is_redblack tr Black h →
             is_redblack (Node Black tl n tr) c (S h).
```

Listing 5.2: Red-black invariant

The definition takes tree parameters, the tree, a natural number and a color. The natural number represents the *black-height* i.e. the number of black nodes in any path from the root node to the leaves. The color represents the *color-context* i.e. the color that the parent node can have in order for the tree to be a sub-tree of a well-formed red-black tree. A leaf is a well-formed red-black tree with black-height 0 in any color-context. A tree with a red root is a well-formed red-black tree with black-height `h` if both of its subtrees have black-height equal to `h` and the color-context is black. Intuitively that means that a red node does not change the black-height and cannot have a red parent. A tree with a red root is a well-formed red-black tree with black-height `h+1` if both of its subtrees have black-height equal to `h` and in whatever color-context. Intuitively that means that a black node increases the black-height by one and it has no restrictions for the color of its parent node. In each case the black-height of the child nodes must be equal in order for the third bullet of the invariant to be satisfied. A tree t is satisfies the red-black tree invariant if

$$\exists\, h,\ \texttt{is\_redblack}\ h\ \texttt{Red}\ t$$

Since the root has to be always black the outermost color-context has to be red in order to enforce this.

We implement an `insert` functions that adds a node to a given tree. The exact implementation of the `insert` function is beyond the scope if this case study and it can be found in the given references. A correct `insert` implementation must preserve the red-black invariant. We can formalize our claim with the following proposition. The corresponding Coq code can be seen in listing 5.3.

$$\forall\, t\ n\ h,\ \texttt{is\_redblack}\ h\ \texttt{Red}\ t\ \rightarrow\ \exists\, h',\ \texttt{is\_redblack}\ h'\ \texttt{Red}\ (\texttt{insert}\ x\ t)$$

```
Definition insert_is_redblack :=
  ∀ x s h, is_redblack s Red h →
                ∃ h', is_redblack (insert x s) Red h'.
```

Listing 5.3: Invariant preservation by `insert`

5.1.3 Generators

In order to be able to test that `insert` preserve the invariant we need a generator for red black trees. We could write a generator that does not enforce the red-black invariant and then filter out the generated trees that are do not respect the invariant. However, this would be extremely inefficient as is highly unlikely that a randomly generated tree will be well-formed. We will write a generator that satisfies the red-black invariant by construction.

First, we write a generator for colors. The `genColor` generator generates red or black with equal probability. Then we write a red-black tree generator that is parameterized by the color-context and the black-height. The generation strategy goes as follows: It the black-height is zero and the color-context is red then only leaf, that are black by default, can be generated. If the black-height is zero and the color-context is black either a leaf or a red node with two leaves for child nodes can be generated. If the black-height is $h+1$ and the color-context is red then when we generate a black node with a randomly key and we call the generator recursively to generate two subtrees with black-height h and color-context red. If the black-height is $h+1$ and the color-context is black that we randomly choose the color of the node. If we pick black then we follow then we do the procedure we just described. If we pick red then we generate a red node with random key and we attach each one two black nodes with random keys and we call recursively the generator to generate for subtrees with black color-context and a black-height of h which we attach in the black nodes.

We can now use this generator to write a red-black tree generator for an arbitrary black-height. We set the outer color-context to red to enforce the root and we call `sized` in order to make the black-height parameter to depend from the global size parameter. We claim that this generator can generate all the possible trees that satisfy the red-black invariant and only them.

Indeed, we can use our framework to verify that claim. First, we prove that `genColor` generates both `Black` and `Red`. Although this is trivial we will use this proof to build the proof for the red-black tree generator in order to avoid reasoning about `genColor` each time it occurs in the proof.

Next we prove that for all possible color-contexts c and black-heights h the generator `genRBTree_height h c` generates all the possible trees that satisfy the red-black invariant with black-height h and color-context c and only them. This lemma can be seen in listing 5.6.

Finally, we can use the previous lemma to prove that `genRBTRee` generates all the possible trees that satisfy the red-black invariant.

5.1.4 Executable Definitions

In order to be able to test the hypothesis about `insert` function we need an executable variant of the `is_redblack` invariant. Since the definition of the invariant we have gives us a `Prop` we are only able to prove that a tree satisfies it rather than efficiently test it. For this reason we will write a boolean predicate that given a tree and a color-context returns `true` if the tree is a well-formed red-black tree under the given color-context. We can describe a decision procedure

```
Section Generators.
  Context {Gen : Type → Type}
          {H: GenMonad Gen}.


  Definition genColor := elements Red [Red; Black].

  Fixpoint genRBTree_height (h : nat) (c : color) :=
    match h with
      | 0 ⇒
        match c with
          | Red ⇒ returnGen Leaf
          | Black ⇒ oneof (returnGen Leaf)
                          [returnGen Leaf;
                            bindGen arbitraryNat (fun n ⇒
                            returnGen (Node Red Leaf n Leaf))]
        end
      | S h ⇒
        match c with
          | Red ⇒
            bindGen (genRBTree_height h Black) (fun t1 ⇒
            bindGen (genRBTree_height h Black) (fun t2 ⇒
            bindGen arbitraryNat (fun n ⇒
            returnGen (Node Black t1 n t2))))
          | Black ⇒
            bindGen genColor (fun c' ⇒
            match c' with
              | Red ⇒
                bindGen (genRBTree_height h Black) (fun tl1 ⇒
                bindGen (genRBTree_height h Black) (fun tl2 ⇒
                bindGen (genRBTree_height h Black) (fun tr1 ⇒
                bindGen (genRBTree_height h Black) (fun tr2 ⇒
                bindGen arbitraryNat (fun n ⇒
                bindGen arbitraryNat (fun nl ⇒
                bindGen arbitraryNat (fun nr ⇒
                returnGen (Node Red (Node Black tl1 nl tr1) n
                                (Node Black tl2 nr tr2))))))))))
              | Black ⇒
                bindGen (genRBTree_height h Black) (fun t1 ⇒
                bindGen (genRBTree_height h Black) (fun t2 ⇒
                bindGen arbitraryNat (fun n ⇒
                returnGen (Node Black t1 n t2))))
            end)
        end
    end.

  Definition genRBTree := sized (fun h ⇒ genRBTree_height h Red).

End Generators.
```

Listing 5.4: Red-black tree generation

```
Lemma genColor_correct:
  genColor ⟷ (fun _ ⇒ True).
```

Listing 5.5: Correctness lemma for `genColor`

```
Lemma genRBTree_height_correct:
  ∀ c h,
    (genRBTree_height h c) ⟷ (fun t ⇒ is_redblack t c h).
```

Listing 5.6: Correctness lemma for `genRBTree_height`

```
Lemma genRBTree_correct:
  genRBTree ⟷ (fun t ⇒ ∃ h, is_redblack t Red h).
```

Listing 5.7: Correctness lemma for `genRBTree`

that can determine whether a tree is well-formed. If the tree is a Leaf then it is well-formed regardless of the color-context. If the root node is black then the tree is well-formed regardless the color-context if both of the subtrees have the same black-height and they are well-formed under black color-context. If the root node is red then the tree is well-formed only under a red color-context if both of the subtrees have the same black-height and they are well-formed under red color-context. In order to express this as a boolean predicate we need a way to calculate the black-height of a tree.

We define the function `black_height_dec` that returns `Some n` if the black-height invariant holds for each node and the black-height of the root node is `n` and `None` if the black-height invariant does not hold. This is depicted in listing 5.8.

```
Fixpoint black_height_dec (t: tree) : option nat :=
  match t with
    | Leaf ⇒ Some 0
    | Node c tl _ tr ⇒
      let h1 := black_height_dec tl in
      let h2 := black_height_dec tr in
      match h1, h2 with
        | Some n1, Some n2 ⇒
          if n1 == n2 then
            match c with
              | Black ⇒ Some (S n1)
              | Red ⇒ Some n1
            end
          else None
        | _, _ ⇒ None
      end
  end.
```

Listing 5.8: A procedure that calculates the black-height

Using the above function we can define a boolean predicate that checks if a red-black tree is well-formed under a color-context.

```
Fixpoint is_redblack_dec (t : tree) (c: color) : bool :=
  match t with
    | Leaf ⇒ true
    | Node c' tl _ tr ⇒
      match c' with
        | Black ⇒
          (black_height_dec tl == black_height_dec tr) &&
          is_redblack_dec tl Black && is_redblack_dec tr Black
        | Red ⇒
          match c with
            | Black ⇒
              (black_height_dec tl == black_height_dec tr) &&
              is_redblack_dec tl Red && is_redblack_dec tr Red
            | Red ⇒ false
          end
      end
  end.
```

Listing 5.9: A procedure that decides if a red-black tree is well-formed under a given color-context

We can now form a checker that check that if we insert a node to a tree that satisfy the red-black invariant the resulting tree will also satisfy the red-black invariant.

```
Section Checker.
  Context {Gen : Type → Type}
          {H: GenMonad Gen}.

  Definition insert_is_redblack_checker : Gen QProp :=
    forAll show_nat arbitraryNat (fun n ⇒
    (forAll showRBTree genRBTree (fun t ⇒
    (is_redblack_dec t Red ==>
     is_redblack_dec (insert n t) Red) : Gen QProp)) : Gen QProp).

End Checker.
```

Listing 5.10: Invariant preservation checker

However, in order to be sure that our decision procedure is correct we have to prove them equivalent to the inductive one. We will use the SSREFLECT[11, 10] library and in particular the inductive predicate `reflect` that relates equivalent boolean and logical definitions. Our goal is to prove that `is_redblack_dec t c` is true if and only if exists a black-height `h` such that `is_redblackc t c h`.

First we prove that if `is_redblackc t c h` holds if and only if `is_redblack_dec t c` is true and `black_height_dec` is equal to `Some n`. Then, we prove that if `is_redblack_dec t c` is true then *t* has a black-height. Using those two lemmas we can prove the final lemma that corresponds to our claim. We can see the statements of the lemmas in listings 5.11 to 5.13.

```
Lemma is_redblackP :
  ∀ (t : tree) (c : color) n,
    reflect (is_redblack t c n)
            (is_redblack_dec t c && (black_height_dec t == Some n)).
```

Listing 5.11: Reflection proof between `is_redblack` and `is_redblack_dec`

```
Lemma has_black_height_dec :
  ∀ t c, is_redblack_dec t c → ∃ n, black_height_dec t = Some n.
```

Listing 5.12: Lemma for `black_height_dec`

```
Lemma is_redblack_exP :
  ∀ (t : tree) (c : color),
    reflect (∃ n, is_redblack t c n)
            (is_redblack_dec t c).
```

Listing 5.13: Reflection proof between `is_redblack` and `is_redblack_dec`

5.1.5 An End to End Proof

The final goal is to prove that the checker we wrote indeed correspond to the specification shown in listing 5.3. We will use `semProperty` to map our checker to a proposition and the prove this equivalent with the `insert_is_redblack` proposition. In order to prove that lemma we make use of the correctness proof for red-black tree generators, the correctness proofs of the executable definitions and finally the proofs for the `Property` combinators we use that are provided by the lemma library of our framework. The formulation of the lemma we proved can be seen in listing 5.14.

```
Lemma insert_is_redblack_checker_correct:
  semProperty insert_is_redblack_checker ↔ insert_is_redblack.
```

Listing 5.14: Lemma stating that the conjecture under test corresponds to the high-level specification

5.2 IFC Case Study

We applied our methodology to verify the generators used in a complex testing infrastructure aimed to test an information flow control machine[1] [12]. The goal was to test that starting from any pair of indistinguishable states any two executions results to final states are also indistinguishable. Each state consists of the instruction and the data memory, a program counter, a stack and a set of registers.

The generators we verified were used to generate state variations, i.e. pairs of indistinguishable states according to a given indistinguishability definition. According to the generation strategy

[1]The development, along with our proofs, can be found at `https://github.com/QuickChick/IFC`

followed the fist state was generated arbitrarily and the second by varying the fist in order to create a second indistinguishable state. The are two kinds of generators, generators that are used to generate arbitrary state components and generators that given a state component produce a variation of it. Those generators were composed in order to write the final generator used for state variation generation. The utmost goal of this testing method was to create only indistinguishable states and all the possible meaningful pairs.

We verified each of these generators with respect to a high-level specification. We proved soundness of the generation strategy, i.e. that any pair generated by the variation generators was indeed indistinguishable, thus state variation generation is sound with respect to indistinguishability. We also prove completeness for the generators with respect to indistinguishability and some additional constraints that arise from the way we are generating the state components. This indicates that there are pairs of indistinguishable states that cannot be generated and this could potentially affect the effectiveness of the method. One could prove the testing method complete by either removing the additional constraint or proving that these constraints do not affect the generality of the method, i.e. if the desired specification hold for states that have these constraints then it holds for all the states.

Through this case study we were able to verify preexisting code that was not written with verification in mind. The generators of this case study were built in a compositional manner that allowed us to structure our proofs in a compositional and modular way. We were able to locate incomplete generators with respect to the indistinguishable relation and we were able to reason about the exact sets of values that can be generated.

Chapter 6

Related Work

6.1 Property Based Testing and Proof Assistants

6.1.1 Isabelle/HOL

A lot of progress has been made in combining random testing and Isabelle theorem prover, by extending the prover with a QuickCkeck-like tool that supports random testing for specifications written in executable fragment of Isabelle/HOL [4, 5, 6]. The New QuickCheck for Isabelle also provides automatic test data generator synthesis taking specifications into account. However, up to our knowledge, no formal guarantee about the completeness and the soundness of the synthesized generators is provided.

6.1.2 Agda/Alfa

The proof assistant Agda/Alfa is extended with a QuickCheck-like tool that facilitates debugging of programs and specifications before a proof is attempted [8]. The are showing how the fact the generators are themselves written in Agda/Alfa can be exploited to use the dependent type system in order to prove properties such us surjectivity about them. For instance, in order to prove surjectivity (i.e. that all the possible elements can indeed be generated) one has to prove that for each the value it exists a random seed such that given that random seed the generator generates that value. Although Dybjer et al. touch the subject of generator completeness they do no not provide any infrastructure for reasoning about the correctness of generators.

6.2 Testing Evaluation Techniques

6.2.1 Mutation Testing

Mutation testing is a method for evaluating the quality of the testing methodology [14] by modifying the software under multiple times in order to create a set of *mutants* that are then being tested with the same methodology as the original program. The percentage of the mutants that are discovered by testing to have different behavior than the original program provides a measure for the effectiveness of the applied testing method. Mutation testing has been traditionally used for evaluating the adequacy of a test suite but it has also been used to spot weaknesses in specifications used for property base testing [16]. Although mutation testing can be effective in evaluating testing methodologies and especially novel test techniques, it is associated with the high cost of finding the inadequacies of the testing method from mutants that appear to have the same behavior as the original program. The effectiveness of mutation testing is also very sensitive to the set of mutation operators applied thus with an inadequate set of mutation operators even a good mutation score can provide very little assurance about the quality of the testing method. Concluding, mutation testing is a technique that can be effective when designed correctly but comes with no formal guarantees about the effectiveness of the testing method that is being evaluated.

6.2.2 Coverage Analysis

Code coverage is also used as a metric to determine the thoroughness of a test suite. Different kinds of coverage can be used in order to evaluate the completeness of the test data, such as statement or branch coverage. This technique can be very effective in finding paths of the programs that have never been exercised and may contain bugs. Nevertheless, even if the code is 100% covered the testing can still be incomplete as test cases that produce other kinds of errors, such as arithmetic overflows may never be produced.

Chapter 7

Conclusions and Future Work

We extended QuickChick with a verification mechanism that allows us to reason about the effectiveness of testing infrastructure. We provide an automatic way to map checkers into logical propositions that correspond to the exact conjecture under test. The user can use those propositions in order to prove that they are equivalent to a more intuitive high-level declarative specification. Our framework facilitates reasoning about probabilistic programs by abstracting over the random generation representation and over-approximating the underlying probability distribution with the set of values that have non-zero probability to be generated. By mapping generators to sets of outcomes one can prove soundness and completeness for generators.

We used our verification methodology in order to prove statements for an already existing complex testing infrastructure with minimal changes to the code. Essentially the only change we had to do was to put generator and checker definitions in a Coq section in order to make them parametric in the generator type constructor. Our framework encourages the user to structure the proofs in a compositional way achieving modularity and thus more robust and scalable proofs. We can build proofs about complex generators that are independent from the implementation of the individual components being used and only depend from their specifications.

We also proved high-level specifications for QuickChick combinators. These lemmas can be used in order to prove statements about user defined generators that are using the combinators without having to unfold their definitions. The verification with respect to high-level specification of built-in combinators is a first stew towards a fully verified QuickChick implementation.

As a future goal, we could try to lift our assumptions by proving that the primitive generator combinators indeed correspond the to set of outcomes semantics we axiomatically give them. This way we can obtain a fully verified QuickChick interface for building generators and checkers. However, in order to be able to to reason about the primitive combinators we should be able to reason about things that are not currently implemented in Coq. Before attempting to prove that primitive combinators are sound and complete with respect to the sets of outcomes that we map them to, QuickChick should be deeper integrated into Coq as currently we are relying on the extraction language for generation of some built-in types and seed generation and passing.

The set of outcomes abstraction provides only an over-approximation of the underlying probability distribution and even generators that are proved correct can have a bad probability distribution that affects the effectiveness of the testing method, as elements which belong to the set of outcomes may have very small probability to be generated in practice. We believe that we can reason about the probability distributions of generators by instantiating the abstract generator type class with a probability monad in order to be able to map generators to probability distributions semantics. This will give us the ability to prove properties about the probability distributions of the generators such as uniformity. To this direction, we might consider using ALEA [19] library, that will allows us to reason about randomized programs in Coq.

Although our tool eases the verification required there is manual effort required to prove the automatically derived propositions equivalent to the desired ones. A potential direction would be to try to provide more automation that could minimize the manual verification effort required. To this end we could build a framework for automatic generator synthesis from specifications and eliminate the manual effort required for required for proving correct custom generators by verifying, using out verification framework, that the automatically produced generators are correct and complete with respect to the given specifications using.

Bibliography

[1] C. Amaral, M. Florido, and V. Santos Costa. PrologCheck – property-based testing in Prolog. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 1–17. Springer International Publishing, 2014.

[2] A. W. Appel. Efficient verified red-black trees, 2011.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. 2006.

[4] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM)*. 2004.

[5] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *2nd International Conference on Certified Programs and Proofs (CPP)*. 2012.

[6] L. Bulwahn. Smart testing of functional programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2012.

[7] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2000.

[8] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 2003.

[9] A. Gill and C. Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. 2007.

[10] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010. RR-7392 RR-7392.

[11] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[12] C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013.

[13] J. Hughes. QuickCheck testing for fun and profit. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*. 2007.

[14] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[15] D. Le, M. A. Alipour, R. Gopinath, and A. Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. Online Draft; Under Submission, 2014.

[16] D. Le, M. A. Alipour, R. Gopinath, and A. Groce. Mutation testing of functional programming languages. Technical report, Oregon State University, 2014.

[17] C. Okasaki. Red-black trees in a functional setting. *J. Funct. Program.*, 9(4):471–477, 1999.

[18] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*. 2011.

[19] C. Paulin-Mohring. Alea: A library for reasoning on randomized algorithms in Coq version 7. Description of a Coq contribution, Université Paris Sud, 2012. Contributions by David Baelde and Pierre Courtieu.

[20] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test*. 2006.

List of Listings

List of Theorems and Definitions