Foundational Property-Based Testing

Zoe Paraskevopoulou[1,2] Cătălin Hriţcu[1] Maxime Dénès[1]
Leonidas Lampropoulos[3] Benjamin C. Pierce[3]

[1]Inria Paris-Rocquencourt [2]ENS Cachan [3]University of Pennsylvania

Abstract Integrating property-based testing with a proof assistant creates an interesting opportunity: reusable or tricky testing code can be formally verified using the proof assistant itself. In this work we introduce a novel methodology for formally verified property-based testing and implement it as a foundational verification framework for QuickChick, a port of QuickCheck to Coq. Our framework enables one to verify that the executable testing code is testing the right Coq property. To make verification tractable, we provide a systematic way for reasoning about the set of outcomes a random data generator can produce with non-zero probability, while abstracting away from the actual probabilities. Our framework is firmly grounded in a fully verified implementation of QuickChick itself, using the same underlying verification methodology. We also apply this methodology to a complex case study on testing an information-flow control abstract machine, demonstrating that our verification methodology is modular and scalable and that it requires minimal changes to existing code.

1 Introduction

Property-based testing (PBT) allows programmers to capture informal conjectures about their code as executable specifications and to thoroughly test these conjectures on a large number of inputs, usually randomly generated. When a counterexample is found it is shrunk to a minimal one, which is displayed to the programmer. The original Haskell QuickCheck [19], the first popular PBT tool, has inspired ports to every mainstream programming language and led to a growing body of continuing research [6,18,26,35,39] and industrial interest [2,33]. PBT has also been integrated into proof assistants [5,10,15,23–25,37] as a way of reducing the cost of formal verification, finding bugs earlier in the verification process and decreasing the number of failed proof attempts: *Testing helps proving!* Motivated by these earlier successes, we have ported the QuickCheck framework to Coq, resulting in a prototype Coq plugin called QuickChick. With QuickChick, we hope to make testing a convenient aid during Coq proof development.

In this paper we explore a rather different way that testing and proving can cooperate in a proof assistant. Since our testing code (and most of QuickChick itself) is written in Coq, we can also formally verify this code using Coq. That is, *proving helps testing*! This verified-testing idea was first proposed a decade ago by Dybjer, Haiyan, and Takeyama [23,24,30] in the context of Agda/Alfa, but it has apparently received very little attention since then [8,9,13].

Why would one want verified testing? Because PBT is very rarely a push-button process. While frameworks such as QuickCheck provide generic infrastructure for writing testing code, it is normally up to the user to compose the QuickCheck combinators in creative ways to obtain effective testing code for the properties they care about. This testing code can be highly non-trivial, so mistakes are hard to avoid. Some types of mistakes are easily detected by the testing itself, while others are not: inadvertently testing a stronger property will usually fail with a counter-example that one can manually inspect, but testing a weaker or just a different property can succeed although the artifact under test is completely broken with respect to the property of interest. Thus, while PBT is effective at quickly finding bugs in formal artifacts, errors in testing code can conceal important bugs, instilling a false sense of confidence until late in the verification process and reducing the benefits of testing.

One response to this problem is providing more automation. QuickCheck uses type classes for this purpose, and other tools go much further—using, for instance, techniques inspired by functional logic programming and constraint solving [10–12, 14, 16, 18, 22, 26, 27]. While automation reduces user effort by handling easy but tedious and repetitive tasks, we are doubtful that the creative parts of writing effective testing code can be fully automated in general (any more than writing non-trivial programs in any other domain can); our experience shows that the parts that cannot be automated are usually tricky enough to also contain bugs [31]. Moreover, the more sophisticated the testing framework becomes, the higher the chances that it is going to contain bugs itself. Given the randomized nature of QuickCheck-style PBT, such bugs can go unnoticed for a long time; for example, it took more than a decade to discover that Haskell's "splittable pseudo-random number generator" was broken [20].

Thus, both for tricky user code and for reusable framework code, verified testing may be an attractive solution. In particular, formal verification allows one to show that *non-trivial testing code is actually testing the intended property*. To make this process viable in practice, we need a modular and scalable way of reasoning about probabilistic testing code. Moreover, for high assurance, we desire a verification framework based on strong formal foundations, with precisely identified assumptions. More speculatively, such a foundational verification framework could serve as a target for certificate-producing metaprograms and external tools that produce testing code automatically (e.g., from inductive type specifications or boolean predicates).

Contributions. We introduce a novel methodology for formally verified PBT and implement it as a foundational verification framework for our QuickChick Coq plugin. To make verification tractable, we provide a systematic way for reasoning about the set of outcomes a random data generator can produce with non-zero probability, abstracting away from actual probabilities. This possibilistic abstraction is not only convenient in practice, but also very simple and intuitive. Beyond this abstraction, our framework is firmly grounded on a fully verified implementation of QuickChick itself. We are the first to successfully verify a QuickCheck-like library—significant validation for our verification methodology. We also describe a significant case study on testing an information-flow control abstract machine. These experimental results are encouraging, indicating that our verification methodology is modular and scalable, requiring minimal changes to existing code. Finally, porting QuickCheck to Coq is a useful side-contribution that is of independent interest, and we hope that we and others will build upon QuickChick in the future. Our verification framework relies on the SSReflect [29] extension to Coq, and is fully integrated into QuickChick, which is available under a permissive open source license at `https://github.com/QuickChick`.

Outline. Our verification framework is illustrated on an example in §2 and presented in full detail in §3. The information-flow case study is discussed in §4. We present related work in §5, before drawing conclusions and discussing future work in §6.

2 Example: Red-Black Trees

In this section we illustrate both the QuickChick plugin for Coq and our verification framework on a simple red-black tree example.[1] A red-black tree is a self-balancing binary search tree in which each non-leaf node stores a data item and is colored either `Red` or `Black` [36]. We define the type of red-black trees of naturals in Coq as follows:

```
Inductive color := Red | Black.
Inductive tree := Leaf : tree | Node : color -> tree -> nat -> tree -> tree.
```

Inserting a new element into a red-black tree is non-trivial as it involves re-balancing to preserve the following invariants: (i) the root is black; (ii) all leaves are black; (iii) red nodes have two black children; and (iv) from any given node, all descendant paths to leaves have the same number of black nodes. (For simplicity, we ignore the binary-search-tree property and focus only on balancing here.) If we wanted to prove that an `insert` function of type `nat -> tree -> tree` preserves the red-black tree invariant, we could take inspiration from Appel [1] and express this invariant in declarative form:

```
Inductive is_redblack' : tree -> color -> nat -> Prop :=
| IsRB_leaf: forall c, is_redblack' Leaf c 0
| IsRB_r: forall 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: forall 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).
Definition is_redblack (t:tree) : Prop := exists h, is_redblack' t Red h.
```

The definition uses a helper inductive relation `is_redblack'`, pronounced "is a red-black subtree," with three parameters: (i) a sub-tree; (ii) the *color-context* `c` (i.e., the color of the parent node); and (iii) the *black-height* `h` of the sub-tree (i.e., the number of black nodes in any path from the root of the sub-tree to a leaf). A leaf is a well-formed red-black sub-tree in any color-context and has black-height `0`. A node is a red-black sub-tree if both its child trees are red-black sub-trees and if the color-context is black in case it has a red root. Moreover, the black-height of the children must be equal, and we increase this height by `1` if the root is black. Using this definition we might like to prove in Coq the following property of `insert`:

```
Definition insert_preserves_redblack : Prop :=
  forall x s, is_redblack s -> is_redblack (insert x s).
```

Before starting a proof of this proposition we would like to quickly check that we did not do any mistakes in the definition of `insert` or `is_redblack`. However, the declarative definition of `is_redblack` is not well adapted to efficient testing. Even if we were able to automatically give an executable interpretation to the inductive definition of

[1] The complete code for this example is available at
`https://github.com/QuickChick/QuickChick/tree/master/examples/RedBlack`

`is_redblack'` [3,4,21,41], we would still have to guess the existentially quantified black-height `h`, which would be highly inefficient. So in order to effectively test the `is_redblack` invariant, we first manually devise an efficiently executable version:

```
Definition is_redblack_bool (t : tree) : bool :=
  is_black_balanced t && has_no_red_red Red t.
```

We omit the definitions of the auxiliaries `is_black_balanced` and `has_no_red_red`. While `is_redblack_bool` allows us to check whether a tree is red-black or not, in order to test the invariant of `insert` using QuickChick, we also need a way to generate random trees. We start by devising a generic tree *generator* using the QuickChick combinators:

```
Definition genColor := elems [Red; Black].
Fixpoint genAnyTree_depth (d : nat) : G tree :=
  match d with
    | 0 => returnGen Leaf
    | S d' => freq [(1, returnGen Leaf);
                    (9, liftGen4 Node genColor (genAnyTree_depth d')
                                     arbitrary (genAnyTree_depth d'))]
  end.
Definition genAnyTree : G tree := sized genAnyTree_depth.
```

The `genAnyTree_depth` auxiliary generates trees of a given depth. If the depth is zero, we generate a leaf using `returnGen Leaf`, otherwise we use the `freq` combinator to choose whether to generate a leaf or to generate a color using `genColor`, a natural number using `arbitrary`, the two sub-trees using recursive calls, and put everything together using `liftGen4 Node`. The code illustrates several QuickChick combinators: (i) `elems` chooses a color from a list of colors uniformly at random; (ii) `returnGen` always chooses the same thing, a `Leaf` in this case; (iii) `freq` performs a biased probabilistic choice choosing a generator from a list using user-provided weights (in the example above we generate nodes 9 times more often than leafs); (iv) `liftGen4` takes a function of 4 arguments, here the `Node` constructor, and applies it to the result of 4 other generators; (v) `arbitrary` is a method of the `Arbitrary` type class, which assigns default generators to frequently used types, in this case the naturals; and (vi) `sized` takes a generator parameterized by a size, in this case `genAnyTree_depth`, and produces a non-parameterized generator by iterating over different sizes. Even this naive generator is easy to get wrong: our first take at it did not include the call to `freq` and was thus only generating full trees, which can cause testing to miss interesting bugs.

The final step for testing the `insert` function using this naive generator is combining the `genAnyTree` generator and the `is_redblack` boolean function into a *property checker*—i.e., the testing equivalent of `insert_preserves_redblack`:

```
Definition insert_preserves_redblack_checker (genTree : G tree) : Checker :=
  forAll arbitrary (fun n => forAll genTree (fun t =>
    is_redblack_bool t ==> is_redblack_bool (insert n t))).
```

This uses two checker combinators from QuickChick: (i) `forAll` produces data using a generator and passes it to another checker; and (ii) `c1 ==> c2` takes two checkers `c1` and `c2` and tests that `c2` does not fail when `c1` succeeds. The “==>” operator also remembers the percentage of inputs that do not satisfy the precondition `c1` and thus have to be discarded without checking the conclusion `c2`. In our running example `c1` and `c2` are two boolean expressions that are implicitly promoted to checkers. Now we have something we can test using the QuickChick plugin, using the `QuickChick` command:

```
QuickChick (insert_preserves_redblack_checker genAnyTree).
*** Gave up! Passed only 2415 tests
Discarded: 20000
```

We have a problem: Our naive generator for trees is very unlikely to generate red-black trees, so the premise of `insert_preserves_redblack_checker` is false and thus the property vacuously true 88% of the time.The conclusion is actually tested infrequently, and if we collect statistics about the distribution of data on which it is tested, we see that the size of the trees that pass the very strong precondition is very small: about 95.3% of the trees have 1 node, 4.2% of them have 3 nodes, 0.4% of them have 5 nodes, and only 0.03% of them have 7 or 9 nodes. So we are not yet doing a good job at testing the property. While the generator above is very simple—it could probably even be generated automatically from the definition of `tree` [3, 5, 42]—in order to effectively test the property we need to write a *property-based generator* that *only* produces red-black trees.

```
Program Fixpoint genRBTree_height (hc : nat*color) {wf wf_hc hc} : G tree :=
  match hc with
  | (0, Red) => returnGen Leaf
  | (0, Black) => oneOf [returnGen Leaf;
                    (do! n <- arbitrary; returnGen (Node Red Leaf n Leaf))]
  | (S h, Red) => liftGen4 Node (returnGen Black) (genRBTree_height (h, Black))
                                   arbitrary (genRBTree_height (h, Black))
  | (S h, Black) => do! c’ <- genColor;
                    let h’ := match c’ with Red => S h | Black => h end in
                    liftGen4 Node (returnGen c’) (genRBTree_height (h’, c’))
                                  arbitrary (genRBTree_height (h’, c’)) end.
Definition genRBTree := bindGen arbitrary (fun h => genRBTree_height (h, Red)).
```

The `genRBTree_height` generator produces red-black trees of a given black-height and color-context. For black-height `0`, if the color-context is `Red` it returns a (black) leaf, and if the color-context is `Black` it uses the `oneOf` combinator to select between two generators: one that returns a leaf, and another that returns a `Red` node with leaf children and a random number. The latter uses do notation for bind (“`do! n <- arbitrary; ...`”) in the `G` randomness monad. For black-height larger than `0` and color-context `Red` we always generate a `Black` node (to prevent red-red conflicts) and generate the sub-trees recursively using a smaller black-height. Finally, for black-height larger than `0` and color-context `Black` we have the choice of generating a `Red` or a `Black` node. If we generate a `Red` node the recursive call is done using the same black-length. The function is shown terminating using a lexicographic ordering on the black-height and color-context.

With this new generator we can run 10000 tests on a laptop in less than 9 seconds, of which only 1 second is spent executing the tests. The the rest is spent extracting to OCaml and running the OCaml compiler (the extraction and compilation part could be significantly sped up; this time is also easily amortized for longer running tests):

```
QuickChick (insert_preserves_redblack_checker genRBTree).
+++ OK, passed 10000 tests
```

Moreover, none of the generated trees fails the precondition and the average size of the trees used for testing the conclusion is 940.7 nodes (compared to 1.1 nodes naively!)

In the process of testing, we have, however, written quite a bit of executable testing code—some of it non-trivial, like the generator for red-black trees. How do we know that this code is testing the declarative proposition we started with? Does our generator for red-black trees only produce red-black trees, and even more importantly can it in principle produce *all* red-black trees? Our foundational testing verification framework supports formal answers to these questions. In our framework the semantics of each generator is the set of values that have non-zero probability of being generated. Building on this, we assign a semantics to each checker expressing the logical proposition it tests, abstracting away from computational constraints like space and time as well as the precise probability distributions of the generators it uses. In concrete terms, a function `semChecker` assigns each `Checker` a `Prop`, and a function `semGen` assigns each generator of type `G A` a (non-computable) set of outcomes with Coq type `A -> Prop`.

```
semChecker : Checker -> Prop
semCheckable : forall (C : Type) `{Checkable C}, C -> Prop.
Definition set T := T -> Prop.
Definition set_eq {A} (m1 m2 : set A) := forall (a : A), m1 a <-> m2 a.
Infix "<-->" := set_eq (at level 70, no associativity) : set_scope.
semGen : forall A : Type, G A -> set A
semGenSize : forall A : Type, G A -> nat -> set A
```

Given these, we can prove that a checker `c` tests a declarative proposition `P` by showing that `semChecker c` is logically equivalent with `P`. Similarly, we can prove that a generator `g` produces the set of outcomes `O` by showing that the set `semGen g` is equal to `O`, using the extensional definition of set equality `set_eq` above. Returning to our red-black tree example we can prove the following top-level theorem:

```
Lemma insert_preserves_redblack_checker_correct:
  semChecker (insert_preserves_redblack_checker genRBTree)
  <-> insert_preserves_redblack.
```

The top-level structure of the checker and the declarative proposition is very similar in this case, and our framework provides lemmas about the semantics of `forAll` and "`==>`" that we can use to make the connection (`semCheckable` is just a variant of `semChecker` described further in §3.2):

```
Lemma semForAllSizeMonotonic {A C} `{Show A, Checkable C} (g : G A) (f : A -> C)
    `{SizeMonotonic _ g} `{forall a, SizeMonotonicChecker (checker (f a))} :
  (semChecker (forAll g f) <-> forall (a:A), a \in semGen g -> semCheckable (f a)).

Lemma semImplication {C} `{Checkable C} (c : C) (b : bool) :
  semChecker (b ==> c) <-> (b -> semCheckable c).

Lemma semCheckableBool (b : bool) : semCheckable b <-> b.
```

Using these generic lemmas, we reduce the original equivalence we want to show to the equivalence between `is_redblack` and `is_redblack_bool` (`reflect` is equivalence between a `Prop` and a `bool` in the SSReflect libraries).

```
Lemma is_redblackP t : reflect (is_redblack t) (is_redblack_bool t).
```

Moreover, we need to show that the generator for red-black trees is complete; i.e., they it can generate all red-black trees. We show this via a series of lemmas, including:

```
Lemma semColor : semGen genColor <--> [set : color].
Lemma semGenRBTreeHeight h c :
  semGen (genRBTree_height (h, c)) <--> [set t | is_redblack' t c h ].
```

```
Lemma semRBTree : semGen genRBTree <--> [set t | is_redblack t].
```

The proofs of these custom lemmas rely again on generic lemmas about the QuickChick combinators that they use. We list most the generic lemmas that we used in this proof:

```
Lemma semReturn {A} (x : A) : semGen (returnGen x) <--> [set x].
```

```
Lemma semBindSizeMonotonic {A B} (g : G A) (f : A -> G B)
      '{Hg : SizeMonotonic _ g} '{Hf : forall a, SizeMonotonic (f a)} :
    semGen (bindGen g f) <--> \bigcup_(a in semGen g) semGen (f a).
```

```
Lemma semElems A (x : A) xs : semGen (elems (x ;; xs)) <--> x :: xs.
```

```
Lemma semOneOf A (g0 : G A) (gs : list (G A)) :
  semGen (oneOf (g0 ;; gs))  <--> \bigcup_(g in (g0 :: gs)) semGen g.
```

While the proof of the red-black tree generator still requires manual effort the user only needs to verify the code she wrote, relying on the precise high-level specifications of all combinators she uses (e.g., the lemmas above). Moreover, all proofs are in terms of propositions and sets, not probability distributions or low-level pseudo-randomness. The complete example is around 150 lines of proofs for 236 lines of definitions. While more aggressive automation (e.g., using SMT) could further reduce the effort in the future, we believe that verifying reusable or tricky testing code (like QuickChick itself or the IFC generators from §4) with our framework is already an interesting proposition.

3 Foundational Verification Framework

As the example above illustrates, the main advantage of using our verified testing framework is the ability to carry out abstract (possibilistic) correctness proofs of testing code with respect to the high-level specifications of the QuickChick combinators. But how do we know that those specifications are correct? And what exactly do we mean by “correct”? What does it mean that a property checker is testing the right proposition, or that a generator is in principle able to produce a certain outcome? To answer these questions with high confidence we have verified QuickChick all the way down to a small set of definitions and assumptions. At the base of our formal construction lies our possibilistic semantics of generators (§3.1) and checkers (§3.2), and an idealized interface for a splittable pseudorandom number generator (splittable PRNG, in §3.3). Our possibilistic abstraction allows us to completely avoid complex probabilistic reasoning at all levels, which greatly improves the scalability and ease of use of our methodology. On top of this we verify all the combinators of QuickChick, following the modular structure of the code (§3.4). We provide support for conveniently reasoning about sizes (§3.5) and about generators for functions (§3.6). Our proofs use a small library for reasoning about non-computable sets in point-free style (§3.7).

3.1 Set-of-Outcomes Semantics for Generators In our framework, the semantics of a generator is the set of outcomes it can produce with non-zero probability. We chose this over a more precise abstraction involving probability distributions, because specifying and verifying probabilistic programs is significantly harder than nondeterministic ones.

Our possibilistic semantics is simpler and easier to work with, allowing us to scale up our verification to realistic generators, while still being precise enough to find many bugs in them (§4). Moreover, the possibilistic semantics allows us to directly relate checkers to the declarative propositions they test (§3.2); in a probabilistic setting the obvious way to achieve this is by only looking at the support of the probability distributions, which would be equivalent to what we do, just more complex. Finally, with our set-of-outcomes semantics, set inclusion corresponds exactly to generator surjectivity from previous work on verified testing [23, 24, 30], while bringing significant improvements to proofs via point-free reasoning (§3.7) and allowing us to verify both soundness and completeness.

QuickChick generators are represented internally the same way as a reader monad [34] with two parameters: a size and a random seed [19] (the bind of this monad splits the seed, which is further discussed in §3.3 and §3.4).

```
Inductive G (A:Type) : Type := MkGen : (nat -> RandomSeed -> A) -> G A.
```

```
Definition run {A : Type} (g : G A) := match g with MkGen f => f end.
```

Formally, the semantics of a generator `g` of type `G A` is defined as the set of elements `a` of type `A` for which there exist a size `s` and a random seed `r` with `run g s r = a`.

```
Definition semGenSize {A : Type} (g : G A) (s : nat) : set A :=
  [set a : A | exists r, run g s r = a].
Definition semGen {A : Type} (g : G A) : set A :=
  [set a : A | exists s, a \in semGenSize g s].
```

We also define `semGenSize`, a variant of the semantics that assigns to a generator the outcomes it can produce for a given size. Reasoning about sizes is discussed in §3.5.

3.2 Possibilistic Semantics of Checkers A property checker is an executable routine that expresses a property under test so that is can be checked against a large number of randomly generated inputs. The result of a test can be either successful, when the property holds for a given input, or it may reveal a counterexample. Property checkers have type `Checker` and are essentially generators of testing results.

In our framework the semantics of a checker is a Coq proposition. The proposition obtained from the semantics can then be proved logically equivalent to the desired high-level proposition that we claim to test. More precisely, we map a checker to a proposition that holds if and only if no counterexamples can possibly be generated, i.e., when the property we are testing is always valid for the generators we are using. This can be expressed very naturally in our framework by stating that all the results that belong to the set of outcomes of the checker are successful (remember that checkers are generators), meaning that they do not yield any counterexample. Analogously to generators, we also define `semCheckerSize` that maps the checker to its semantics for a given size.

```
Definition semCheckerSize (c : Checker) (s : nat): Prop :=
  successful @: semGenSize c s \subset [set true].
```

```
Definition semChecker (c : Checker) : Prop := forall s, semCheckerSize c s.
```

Universally quantifying over all sizes in the definition of `semChecker` is a useful idealization. While in practice QuickChick uses an incomplete heuristic for trying out different sizes in an efficient way, it would be very cumbersome and completely unenlightening to reason formally about this heuristic. By deliberately abstracting away from this source of incompleteness in QuickChick, we obtain a cleaner mathematical model. Despite this idealization, it is often not possible to completely abstract away from the sizes in our proofs, but we provide ways to make reasoning about sizes convenient (§3.5).

In order to make writing checkers easier, QuickChick provides the type class `Checkable` that defines `checker`, a coercion from a certain type (e.g., `bool`) to `Checker`. We trivially give semantics to a type that is instance of `Checkable` with:

```
Definition semCheckableSize {A} ‘{Checkable A} (a : A) (s : nat) : Prop :=
  semCheckerSize (checker a) s.
Definition semCheckable {A} ‘{Checkable A} (a : A) : Prop := semChecker (checker a).
```

3.3 Splittable Pseudorandom Number Generator Interface QuickChick's splittable PRNG is written in OCaml. The rest of QuickChick is written and verified in Coq and then extracted to OCaml. Testing happens outside of Coq for efficiency reasons. The random seed type and the low-level operations on it, such as splitting a random seed and generating booleans and numbers, are simply axioms in Coq. Our proofs also assume that the random seed type is inhabited and that the operations producing numbers from seeds respect the provided range. All these axioms would disappear if the splittable PRNG were implemented in Coq. One remaining axiom would stay though, since it is inherent to our idealized model of randomness:

```
Axiom randomSplit : RandomSeed -> RandomSeed * RandomSeed.
Axiom randomSplitAssumption :
  forall s1 s2 : RandomSeed, exists s, randomSplit s = (s1,s2).
```

This axiom says that the `randomSplit` function is surjective. This axiom has non-trivial models: the `RandomSeed` type could be instantiated with the naturals, infinite streams, infinite trees, etc. One can also easily show that, in all non-trivial models of this axiom, `RandomSeed` is an infinite type. In reality though, PRNGs work with finite seeds. Our axiom basically takes us from pseudorandomness to ideal mathematical randomness, as used in probability theory. This idealization seems unavoidable for formal reasoning and it would also be needed even if we did probabilistic as opposed to possibilistic reasoning. Conceptually, one can think of our framework as raising the level of discourse in two ways: (i) idealizing pseudorandomness to probabilities; and (ii) abstracting probability distributions to their support sets. While the abstraction step could be formally justified (although we do not do this at the moment), the idealization step has to be taken on faith and intuition only. We believe that the possibilistic semantics from §3.1 and §3.2 and the axioms described here are simple and intuitive enough to be trusted; together with Coq they form the trusted computing base of our foundational verification framework.

3.4 Verified Testing Combinators QuickChick provides numerous combinators for building generators and checkers. Using the semantics described above, we prove that each of these combinators satisfies a high-level declarative specification. We build our proofs following the modular organization of the QuickChick code (Fig. 1): only a few low-level generator combinators directly access the splittable PRNG interface and the concrete representation of generators. All the other combinators are built on top of the low-level generators. This modular organization is convenient for structuring our proofs all the way down. Table 1 illustrates an important part of the combinator library and how it is divided into low-level generators, high-level generators, and checkers.

The verification of low-level generators has to be done in a very concrete way that involves reasoning about random seeds. However, once we fully specify these generators in terms of their sets of outcomes, the proof of any generator that builds on top of them can be done in a fully compositional way that only depends on the set of outcomes specifications of the combinators used, abstracting away from the internal representation of generators, the implementation of the combinators, and the PRNG interface.

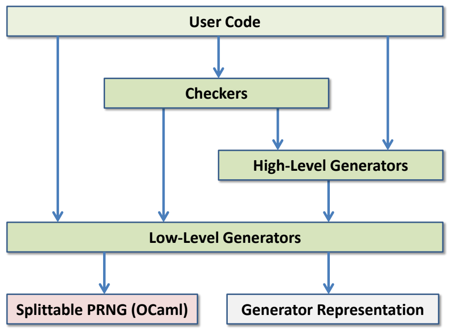

Figure 1. QuickChick organization diagram

Checker Combinators	`implication: forall {C : Type} ‘{Checkable C}, bool -> C -> Checker`

	`shrinking: forall {C A : Type} ‘{Checkable C},`
	`(A -> list A) -> A -> (A -> C) -> Checker`

	`expectFailure: forall {C : Type} ‘{Checkable C}, C -> Checker`

	`forAll: forall {A C : Type} ‘{Show A, Checkable C},`
	`G A -> (A -> C) -> Checker`

	`forAllShrink: forall {A C : Type} ‘{Show A, Checkable C},`
	`G A -> (A -> list A) -> (A -> C) -> Checker`

High-level Generator Combinators	`liftGen: forall {A B : Type}, (A -> B) -> G A -> G B`
	`sequenceGen: forall {A : Type}, list (G A) -> G (list A)`
	`foldGen: forall {A B : Type}, (A -> B -> G A) -> list B -> A -> G A`
	`oneof: forall {A : Type}, G A -> list (G A) -> G A`
	`frequency: forall {A : Type}, G A -> list (nat * G A) -> G A`
	`vectorOf: forall {A : Type}, nat -> G A -> G (list A)`
	`listOf: forall {A : Type}, G A -> G (list A)`
	`elements: forall {A : Type}, A -> list A -> G A`
Low-level Generator Combinators	`bindGen: forall {A B : Type}, G A -> (A -> G B) -> G B`
	`fmap: forall {A B : Type}, (A -> B) -> G A -> G B`
	`sized: forall {A: Type}, (nat -> G A) -> G A`
	`resize: forall {A: Type}, nat -> G A -> G A`
	`suchThatMaybe: forall {A : Type}, G A -> (A -> bool) -> G (option A)`
	`choose: forall {A : Type} ‘{ChoosableFromInterval A}, (A * A) -> G A`

Table 1. Selected QuickChick combinators

As we want the proofs to be structured in compositional way and only depend on the specifications and not the implementation of the combinators, we make the combinator implementations opaque for later proofs by enclosing them in a module that only exports their specifications. The size of the QuickChick framework (excluding examples) is around 2.4 kLOC of definitions and 2.0 kLOC of proofs.

3.5 Conveniently Reasoning about Sizes QuickChick does not prescribe how the generators use the size parameter: some of them are actually *unsized* (they do not use their size parameter at all), while others are *sized* (they produce data depending on the size). For instance `genColor` from §2 is unsized—it always chooses uniformly at random between `Red` or `Black`—while `genAnyTree` and `genRBTree` are both sized. For sized generators, the precise size dependency can vary; indeed, there can be different notions of size for the same type: e.g., for `genAnyTree` size means depth, while for `genRBTree` it means black-height. Finally, some generators take the size parameter to mean the maximal size of the data they generate (e.g., the default generator for naturals, `genAnyTree`, `genRBTree`), while others take it to mean the exact size (e.g., `sized (fun h => genRBTree_height (h, Red))` would be such a generator). Through our verification we discovered that unsized generators and sized generators using maximal size are easier to compose right since they allow stronger principles for compositional reasoning.

In §3.2 we defined the semantics of checkers by universally quantifying over all sizes, so one could naively expect that with this idealization there would be no need to unfold `semGen` and reason explicitly about the size parameter in terms of `semGenSize` in our generator proofs. Unfortunately, this is not always the case: low-level combinators taking several generators (or generator-producing functions) as arguments call all these arguments with the *same* size parameter (reader monad). For instance, `bindGen g f` returns a generator that given a size `s` and a seed `r`, splits `r` into `(r1,r2)`, runs `g` on `s` and `r1` in order to generate a value `v`, then applies `f` to `v` and runs the resulting generator with the same size `s` and with seed `r2`. It would be very tempting to try to give `bindGen` the following very intuitive specification, basically interpreting it as the bind of the nondetederminism monad:[2]

```
  semGen (bindGen g f) <--> \bigcup_(a \in semGen g) semGen (f a).
```

This intuitive specification is, however, wrong in our setting. The set on the right-hand side contains elements that are generated from `(f a)` for some size parameter, whereas `a` is an element that has been generated from `g` with a different size parameter. This would allow us to prove the following generator complete

```
  gAB = bindGen gA (fun a => bindGen gB (fun b => returnGen (a,b)))
```

with respect to `[set : A * B]` for any generators `gA` and `gB` for types `A` and `B`, even in the case when `gA` and `gB` are fixed-size generators, in which case `gAB` only produces pairs of equally-sized elements. In our setting, a correct specification of `bindGen` that works for arbitrary generators can only be given in terms of `semGenSize`, where the size parameter is also threaded through explicitly at the specification level:

```
Lemma semBindSize A B (g : G A) (f : A -> G B) (s : nat) :
  semGenSize (bindGen g f) s <--> \bigcup_(a in semGenSize g s) semGenSize (f a) s.
```

The two calls to `semGenSize` on the right-hand side are now passed the same size.

While in general we cannot avoid explicitly reasoning about the interaction between the ways composed generators use sizes, we can avoid it for two large classes of generators: unsized and size-monotonic generators. We call a generator size-monotonic

[2] Indeed, in a preliminary version of our framework [38] the low-level generators were axiomatized instead of verified with respect to a semantics, and we took this specification as an axiom.

when increasing the size produces a larger (with respect to set inclusion) set of outcomes. Formally, these properties of generators are expressed by the following two type classes:

```
Class Unsized {A} (g : G A) := {
  unsized : forall s1 s2, semGenSize g s1 <--> semGenSize g s2 }.

Class SizeMonotonic {A} (g : G A) := {
  monotonic : forall s1 s2, s1 <= s2 -> semGenSize g s1 \subset semGenSize g s2 }.
```

The `gAB` generator above is in fact complete with respect to `[set : A * B]` if at least one of `gA` and `gB` is `Unsized` or if both `gA` and `gB` are `SizeMonotonic`. We can prove this conveniently using specialized specifications for `bindGen` from our library, such as the `semBindSizeMonotonic` lemma from §2 or the lemma below:

```
Lemma semBindUnsized1 {A B} (g : G A) (f : A -> G B) `{H : Unsized _ g}:
    semGen (bindGen g f) <--> \bigcup_(a in semGen g) semGen (f a).
```

Our library additionally provides generic type-class instances for proving automatically that generators are `Unsized` or `SizeMonotonic`. For instance `Unsized` generators are always `SizeMonotonic` and a bind is `Unsized` when both its parts are `Unsized`:

```
Declare Instance unsizedMonotonic {A} (g : G A) `{Unsized _ g} : SizeMonotonic g.

Declare Instance bindUnsized {A B} (g : G A) (f : A -> G B)
    `{Unsized _ g} `{forall x, Unsized (f x)} : Unsized (bindGen g f).
```

There is a similar situation for checkers, for instance the lemma providing a specification to `forAll` (internally just a `bindGen`) we used in §2 is only correct because of the preconditions that both the generator and the body of the `forAll` are `SizeMonotonic`.

3.6 Verified Generation of Functions In QuickChick we emulate (and verify!) the original QuickCheck's approach to generating functions [17, 19]. In order to generate a function `f` of type `a->b` we use a generator for type `b`, making sure that subsequent calls to `f` with the same argument use the same random seed. Upon function generation, QuickCheck captures a random seed within the returned closure. The closure calls a user-provided `coarbitrary` method that deterministically derives a new seed based on the captured seed and each argument to the function, and then passes this new seed to the generator for type `b`.

Conceptually, repeatedly splitting a random seed gives rise to an infinite binary tree of random seeds. Mapping arguments of type `a` to tree paths gives rise to a natural implementation of the `coarbitrary` method. For random generation to be correct, the set of all possible paths used for generation needs to be prefix-free: if any path is a subpath of another then the values that will be generated will be correlated.

To make our framework easier to use, we decompose verifying completeness of function generation into two parts. On the client side, the user need only provide an injective mapping from the function argument type to positives (binary positive integers) to leverage the guarantees of our framework. On the framework side, we made the split-tree explicit using lists of booleans as paths and proved a completeness theorem:

```
Theorem SplitPathCompleteness (l : list SplitPath) (f : SplitPath -> RandomSeed) :
  PrefixFree l -> exists (s : RandomSeed), forall p, In p l -> varySeed p s = f p.
```

Intuitively, given any finite prefix-free set of paths S and a function f from paths to seeds, there exists a random seed s such that following any path p from S in s's split-tree, we get f p. In addition, our framework provides a mapping from Coq's positives to a prefix-free subset of paths. Combining all of the above with the user-provided injective mapping to positives, the user can get strong correctness guarantees for function generation.

```
Theorem arbFunComplete `{CoArbitrary A, Arbitrary B} (max:A) (f:A-> B) (s:nat) :
  s = Pos.to_nat (coarbitrary max) -> (semGenSize arbitrary s <--> setT) ->
  exists seed, forall a, coarbLe a max -> run arbitrary s seed a = f a.
```

For generators for finite argument function types, the above is a full completeness proof, assuming the result type also admits a complete generator. For functions with infinite argument types we get a weaker notion of completeness: given any finite subset A of a and any function f : a->b, there exists a seed that generates a function f' that agrees with f in A. We have a paper proof (not yet formalized) extending this to a proof of completeness for arbitrary functions using transfinite induction and assuming the set of seeds is uncountable.

3.7 Reasoning about non-computable sets Our framework provides convenient ways of reasoning about the set-of-outcomes semantics from §3.1. In particular, we favor as much as possible point-free reasoning by relating generator combinators to set operations. To this aim, we designed a small library for reasoning about non-computable sets that could be generally useful. A set A over type T is represented by a function P : T -> Prop such that P x expresses whether x belongs to A. On such sets, we defined and proved properties of (extensional) equality, inclusion, union and intersection, product sets, iterated union and the image of a set through a function. Interestingly enough, we did not need the set complement, which made it possible to avoid classical logic. Finally, in Coq's logic, extensional equality of predicates does not coincide with the primitive notion of equality. So in order to be able to rewrite with identities between sets (critical for point-free reasoning), we could have assumed some extensionality axioms. However, we decided to avoid this and instead used generalized rewriting [40], which extends the usual facilities for rewriting with primitive equality to more general relations.

4 Case Study: Testing Noninterference

We applied our methodology to verify the existing generators used in a complex testing infrastructure for information flow control machines [31, 32]. The machines dynamically enforce noninterference: starting from any pair of indistinguishable states, any two executions result in final states are also indistinguishable. Instead of testing this end-to-end property directly we test a stronger single-step invariant proposed in [31, 32](usually called unwinding conditions [28]). Each generated machine state consists of instruction and data memories, a program counter, a stack and a set of registers. The generators we verified produce pairs of indistinguishable states according to a certain indistinguishability definition. The first state is generated arbitrarily and the second is produced by randomly varying the first in order to create an indistinguishable state.

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 proved completeness of the generators with respect to a set of outcomes that is smaller than all possible indistinguishable states, precisely capturing the behavior of our generators. While generating all pairs of indistinguishable states seems good in theory, in practice it is more efficient to bound the size of the generated artifacts. However, the trade-off between completeness and efficiency needs to be considered carefully and our framework allowed us to understand and precisely characterize what additional constraints we enforce in our generation, revealing a number of bugs in the process. One of the trade-offs we had to precisely characterize in our specs is that we only generate instruction sequences of length 2, since we are only going to execute at most one instruction in a single step, but we must also allow the program counter to point to different instructions. This greatly improves efficiency since it is much cheaper to generate a couple than an entire sequence of reasonable instructions.

In some cases, we were not able to prove completeness with respect to the specification we had in mind when writing the generators. These cases revealed bugs in our generation that were not found during extensive prior testing and experiments. Some revealed hidden assumptions we had made while testing progressively more complex machines. For instance, in the simple stack-machine from [31], the label of the saved program counters on the stack was always decreasing. When porting the generators to more complex machines that invalidated this assumption, one should have restructured generation to reflect this. In our attempts to prove completeness this assumption surfaced and we were able to fix the problem, validate our fixes and see a noticeable improvement in bug-finding capabilities (some of the bugs we introduced on purpose in the IFC machine to evaluate testing were found much faster).

Other bugs we found were minor errors in the generation. For instance, when generating an indistinguishable atom from an existing one, most of the time we want to preserve the type of the atom (pointers to pointers, labels to labels, etc.) while varying the payload. This is necessary for example in cases where the atoms will be touched by the same instruction that expects a pointer at a location and finding something else there would raise a non-informative (for IFC) exception. On the other hand we did not *always* want to generate atoms of the same type, because some bugs might only be exposed in those cases. We had forgotten to vary the type in our generation which was revealed and fixed during proving. Fixing all these completeness bugs had little impact on generator efficiency, while giving us better testing.

In this case study we were able to verify existing code that was not written with verification in mind. For verifying $\approx$2000 lines of Coq code (of which around $\approx$1000 lines deal strictly with generation and indistinguishability and the other $\approx$1000 lines are transitively used definitions) our proofs required $\approx$2000 lines of code. We think this number could be further reduced in the future by taking better advantage of point-free reasoning and the non-computable sets library. With minimal changes to the existing code (e.g. fixing revealed bugs) we were able to structure our proofs in a compositional and modular way. We were able to locate incomplete generators with respect to our specification, reason about the exact sets of values that they can generate, and fix real problems in an already thoroughly tested code base.

5 Related Work

In their seminal work on combining testing and proving in dependent type theory [23,24], Dybjer *et al.*, also introduce the idea of verifying generators and identify surjectivity (completeness) as the most important property to verify. They model generators in Agda/Alfa as functions transforming finite binary trees of naturals to elements of the domain, and prove from first principles the surjectivity of several generators similar in complexity to our red-black tree example generator from §2. They study a more complex propositional solver using a randomized Prolog-like search [24, 30], but apparently only prove this generator correct informally, on paper [30, Appendix of Chapter 4]. The binary trees of naturals are randomly generated outside the system, and roughly correspond both to our seed and size. While Dybjer *et al.*'s idea of verifying generators is pioneering, we take this further and build a generic verification framework for PBT. By separating seeds and sizes, as already done in QuickCheck [19], we get much more control over the size of the data we can construct. While this makes formal verification a bit more difficult as we have to explicitly reason about sizes in our proofs, we support compositional size reasoning via type classes such as `Unsized` and `SizeMonotonic` (§3.5). Finally, our checkers do not have a fixed shape, but are also built and verified in a modular way.

In a previous attempt at bringing PBT to Coq, Wilson [42] created a simplified QuickCheck like tool for automatically generating test inputs for a small class of testable properties. His goal was to support dependently typed programming in Coq with both proof automation and testing support. In the same work, attempts are made to aid proof automation by disproving false generalizations using testing. However there is no support for writing generations in Coq and therefore also no way of proving interesting facts about generators. In addition, the generation is executed inside Coq which can lead to inefficiency issues without proper care. For example, as they report, a simple multiplication `200 x 200` takes them 0.35s, while at the same time our framework can generate and test the insert property on around 400 red-black trees (§2).

A different approach at producing a formalized testing tool was taken in the context of FocalTest [13]. Their verification goal is different; they want to provide a fully verified constraint-based testing tool that automatically generates MC/DC compliant test suites from high-level specifications. They prove a translation from their high level ML-like language to their constraint solving language.

Isabelle provides significant support for testing, in particular via a push-button testing framework [5, 10]. The current goals for QuickChick are different: we do not try to automatically generate test data satisfying complex invariant, but provide ways for the users to construct property-based generators. Both of these approaches have their advantages: the automatic generation of random test data in Isabelle is relatively easy to use for novices, while the approach taken by QuickChick gives the experienced user more control over how the data is generated. In the future, it would make sense to combine these approaches and obtain the best of both worlds.

A work perhaps closer related to ours, but still complementary, is the one by Brucker et al [9], who in their HOL-TestGen framework also take a more foundational approach to testing methodologies, making certain assumptions explicit. Instead of using adequacy criteria like MC/DC [13], they provide feedback on "what remains to be proved" after testing. This is somewhat similar in concept to the notion of completeness of generators in our framework. However the tool's approach to generation is automatic small-scale exhaustive testing with no support for manual generators. Our experience is that randomized testing with large instances scales much better in practice. A more recent paper on HOL-TestGen [7] presents a complete case study and establishes a formal correspondence between the specifications of the program under test and the properties that will be tested after optimization.

6 Conclusion and Future Work

We introduce a novel methodology for formally verified PBT and implement it as a foundational verification framework for QuickChick, our Coq clone of Haskell QuickCheck. Our verification framework is firmly grounded in a verified implementation of QuickChick itself. This illustrates an interesting interaction between testing and proving in a proof assistant, showing that proving can help testing. This also reinforces the general idea that testing and proving are synergistic activities, and gives us hope that a virtuous cycle between testing and proving can be achieved in a theorem prover.

Future work. Our framework reduces the effort of proving the correctness of testing code to a reasonable level, so verifying reusable or tricky code should already be an interesting proposition in many cases. The sets of outcomes abstraction also seems well-suited for more aggressive automation in the future (e.g., using an SMT solver).

Maybe more importantly, one should also strive to reduce the cost of effective testing in the first place. For instance, we are working on a property-based generator language in which programs can be interpreted both as boolean predicates and as generators for the same property. Other tools from the literature provide automation for testing [10–12, 14, 16, 18, 22, 26, 27], still, with very few exceptions [13], the code of these tools is fully trusted. While for some of these tools full formal verification might be too ambitious at the moment, having these tools produce certificates that can be checked in a foundational framework like ours seems well within reach.

Acknowledgments. We thank John Hughes for insightful discussions and the anonymous reviewers for their helpful comments.

References

1. A. W. Appel. Efficient verified red-black trees. Manuscript, 2011.
2. T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with Quviq QuickCheck. *Erlang Workshop*, 2006.
3. S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. *TPHOLs*. 2009.
4. S. Berghofer and T. Nipkow. Executing higher order logic. *TYPES*. 2002.
5. S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. *SEFM*. 2004.
6. J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. *ESOP*. 2010.
7. A. D. Brucker, L. Brügger, and B. Wolff. Formal firewall conformance testing: an application of test and proof techniques. *STVR*, 25(1):34–71, 2015.
8. A. D. Brucker and B. Wolff. Interactive testing with HOL-TestGen. In *Proceedings of the 5th International Conference on Formal Approaches to Software Testing*. 2006.
9. A. D. Brucker and B. Wolff. On theorem prover-based testing. *FAC*, 25(5):683–721, 2013.
10. L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. *CPP*. 2012.
11. L. Bulwahn. Smart testing of functional programs in Isabelle. *LPAR*. 2012.

12. L. Bulwahn. *Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming*. PhD thesis, Technische Universität München, 2013.
13. M. Carlier, C. Dubois, and A. Gotlieb. A first step in the design of a formally verified constraint-based testing tool: FocalTest. *TAP*. 2012.
14. M. Carlier, C. Dubois, and A. Gotlieb. FocalTest: A constraint programming approach for property-based testing. *SDT*. 2013.
15. H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios. Integrating testing and interactive theorem proving. *ACL2*, 2011.
16. J. Christiansen and S. Fischer. EasyCheck – test data for free. *FLOPS*. 2008.
17. K. Claessen. Shrinking and showing functions: (functional pearl). *Haskell Symposium*. 2012.
18. K. Claessen, J. Duregård, and M. H. Pałka. Generating constrained random data with uniform distribution. *FLOPS*. 2014.
19. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ICFP*. 2000.
20. K. Claessen and M. Pałka. Splittable pseudorandom number generators using cryptographic hashing. *Haskell Symposium*. 2013.
21. D. Delahaye, C. Dubois, and J.-F. Étienne. Extracting purely functional contents from logical inductive types. *TPHOLs*. 2007.
22. J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. *Haskell Symposium*. 2012.
23. P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. *TPHOLs*. 2003.
24. P. Dybjer, Q. Haiyan, and M. Takeyama. Random generators for dependent types. *ICTAC*. 2004.
25. C. Eastlund. DoubleCheck your theorems. In *ACL2*, 2009.
26. B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. *ESOP*. 2015.
27. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. *PPDP*. 2007.
28. J. A. Goguen and J. Meseguer. Unwinding and inference control. *IEEE S&P*. 1984.
29. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *JFR*, 3(2):95–152, 2010.
30. Q. Haiyan. *Testing and Proving in Dependent Type Theory*. PhD thesis, Chalmers, 2003.
31. C. Hriţcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. *ICFP*. 2013.
32. C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. A. de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. arXiv:1409.0393; Submitted to Special Issue of Journal of Functional Programming for ICFP 2013, 2014.
33. J. Hughes. QuickCheck testing for fun and profit. *PADL*. 2007.
34. M. P. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*, 1995.
35. C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. *POPL*, 2012.
36. C. Okasaki. Red-black trees in a functional setting. *JFP*, 9(4):471–477, 1999.
37. S. Owre. Random testing in PVS. In *Workshop on Automated Formal Methods*, 2006.
38. Z. Paraskevopoulou and C. Hriţcu. A Coq framework for verified property-based testing. Internship Report, Inria Paris-Rocquencourt, 2014.
39. C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. *Haskell Symposium*. 2008.
40. M. Sozeau. A new look at generalized rewriting in type theory. *JFR*, 2(1):41–62, 2009.
41. P.-N. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. *CPP*. 2012.
42. S. Wilson. *Supporting dependently typed functional programming with proof automation and testing*. PhD thesis, The University of Edinburgh, 2011.