Foundational Property Based Testing

ITP, Nanjing 2015

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

August 27, 2015

Testing and Proving

- Proving
  - very expensive, hard
  - strong guarantees

Testing and Proving

- Proving
    - very expensive, hard
    - strong guarantees
- Testing
    - fast, easy
    - weak guarantees

Testing and Proving

- Proving
  - very expensive, hard
  - strong guarantees
- Testing
  - fast, easy
  - weak guarantees
- Testing helps proving!
  - Decreases the cost of formal proofs
  - Many successful projects that integrate testing into a proof assistant (Isabelle/HOL, Adga/Alfa, ...)
  - QuickChic for Coq

Testing and Proving

- Proving
  - very expensive, hard
  - strong guarantees
- Testing
  - fast, easy
  - weak guarantees
- Testing helps proving!
  - Decreases the cost of formal proofs
  - Many successful projects that integrate testing into a proof assistant (Isabelle/HOL, Adga/Alfa, ...)
  - QuickChic for Coq
- Can proving help testing?

Foundational property based testing framework

- Formally verify testing infrastucture
- Does it correspond to the desired property?
- Our framework builts on top of QuickChick, our PTB tool for Coq

Property Based Testing

- Allows to test code in terms of *functional correctness* by generating a large number of randomly generated inputs
- High level of automation

Property Based Testing

- Allows to test code in terms of *functional correctness* by generating a large number of randomly generated inputs
- High level of automation
- The user has to write:

Property Based Testing

- Allows to test code in terms of *functional correctness* by generating a large number of randomly generated inputs
- High level of automation
- The user has to write:
  - **Generators**
    - Random generation of input data

Property Based Testing

- Allows to test code in terms of *functional correctness* by generating a large number of randomly generated inputs
- High level of automation
- The user has to write:
  - **Generators**
    - Random generation of input data
  - **Checkers**
    - programs that test the desired specification

QuickChick

- Property based testing for Coq (port of Haskell's QuickCheck)
- Implemented in Coq
- It relies on extraction to OCaml for efficient execution
- Low level random generation primitives are implemented in OCaml
- Provides a library of combinators that are used to construct *Generators* and *Checkers*

Contributions

- Testing framework: QuickChick

---

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Contributions

- Testing framework: QuickChick
- Verification framework

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Contributions

- Testing framework: QuickChick
- Verification framework
    - We give semantics to both generators and checkers

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Contributions

- Testing framework: QuickChick
- Verification framework
  - We give semantics to both generators and checkers
  - Generators
    - Prove them sound and complete w.r.t to specifications
    - Abstraction: reason about their *set of outcomes*
    - Avoid probabilistic reasoning
    - The idea of verified generators dates back to Dybjer *et al.* [1]

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Contributions

- Testing framework: QuickChick
- Verification framework
  - We give semantics to both generators and checkers
  - Generators
    - Prove them sound and complete w.r.t to specifications
    - Abstraction: reason about their *set of outcomes*
    - Avoid probabilistic reasoning
    - The idea of verified generators dates back to Dybjer *et al.* [1]
  - Checkers
    - Prove that they correspond to the right logical proposition

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Contributions

- Testing framework: QuickChick
- Verification framework
  - We give semantics to both generators and checkers
  - Generators
    - Prove them sound and complete w.r.t to specifications
    - Abstraction: reason about their *set of outcomes*
    - Avoid probabilistic reasoning
    - The idea of verified generators dates back to Dybjer *et al.* [1]
  - Checkers
    - Prove that they correspond to the right logical proposition
  - We give specifications to QuickChick combinators and prove them correct w.r.t them
    - verify the testing tool itself
    - facilitate reasoning about combinators and checkers

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Contributions

- Testing framework: QuickChick
- Verification framework
  - We give semantics to both generators and checkers
  - Generators
    - Prove them sound and complete w.r.t to specifications
    - Abstraction: reason about their *set of outcomes*
    - Avoid probabilistic reasoning
    - The idea of verified generators dates back to Dybjer *et al.* [1]
  - Checkers
    - Prove that they correspond to the right logical proposition
  - We give specifications to QuickChick combinators and prove them correct w.r.t them
    - verify the testing tool itself
    - facilitate reasoning about combinators and checkers
  - **Case studies**: Red-black trees (next), testing noninterference

[1] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. TPHOLs. 2003.

Running Example

Red-black Trees

Red-Black Trees

- Binary trees with an additional color label to each node

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

- Invariant:
  - The root is always black
  - The leaves are empty and black
  - For each node the path to each possible leaf has the same number of black nodes
  - Red nodes can only have black children
- RB tree operations should preserve the invariant
- We want to *test* that for `insert`

Declarative Definitions

- The property to we would like to *test*

```
Definition insert_preserves_rb : Prop :=
  ∀ (x : nat) (t : tree) , is_redblack t → is_redblack (insert x t).
```

Declarative Definitions

- The property to we would like to *test*

```
Definition insert_preserves_rb : Prop :=
  ∀ (x : nat) (t : tree) , is_redblack t → is_redblack (insert x t).
```

- Red-black invariant

```
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).

Definition is_redblack (t: tree) : Prop := ∃ h, is_redblack' t Red h.
```

- But declarative definitions are not well suited to testing

Testing the property

We need...

- An (efficiently) executable definition of the invariant

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

Testing the property

We need...

- An (efficiently) executable definition of the invariant

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

- An arbitrary tree generator

```
Fixpoint genAnyTree_depth (h : nat) : G tree :=
  match h with
    | 0 ⇒ returnGen Leaf
    | S h' ⇒ freq [(1, returnGen Leaf);
                   (9, liftGen4 Node genColor (genAnyTree_depth h')
                                genNat (genAnyTree_depth h'))]
  end.
Definition genAnyTree : G tree := bindGen genNat genAnyTree_depth.
```

Testing the property

We need...

- An (efficiently) executable definition of the invariant

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

- An arbitrary tree generator

```
Fixpoint genAnyTree_depth (h : nat) : G tree :=
  match h with
    | 0 ⇒ returnGen Leaf
    | S h' ⇒ freq [(1, returnGen Leaf);
                   (9, liftGen4 Node genColor (genAnyTree_depth h')
                                 genNat (genAnyTree_depth h'))]
  end.
Definition genAnyTree : G tree := bindGen genNat genAnyTree_depth.
```

- A property checker

```
Definition insert_preserves_rb_checker (genTree : G tree) : Checker :=
  forAll genNat (fun n ⇒ forAll genTree (fun t ⇒
    is_redblack_bool t ==> is_redblack_bool (insert n t))).
```

So, are we done?

```
QuickChick (insert_preserves_rb_checker genAnyTree).

*** Gave up! Passed only 2415 tests
Discarded: 20000
```

So, are we done?

```
QuickChick (insert_preserves_rb_checker genAnyTree).

*** Gave up! Passed only 2415 tests
Discarded: 20000
```

- Our simple generator is very inefficient!
- In QuickChick test cases that do not satisfy the preconditions are discarded
- For most of the test cases the property is vacuously true

```
Definition insert_preserves_rb_checker (genTree : G tree) : Checker :=
  forAll genNat (fun n ⇒ forAll genTree (fun t ⇒
    is_redblack_bool t ==> is_redblack_bool (insert n t))).
```

Can we do better?

```
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))
                                      genNat (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'))
                                 genNat (genRBTree_height (h', c')) end.

Definition genRBTree := bindGen genNat (fun h ⇒ genRBTree_height (h, Red)).
```

- We claim that this generator produces only RB trees.

Can we do better?

```
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))
                                       genNat (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'))
                                  genNat (genRBTree_height (h', c')) end.

Definition genRBTree := bindGen genNat (fun h ⇒ genRBTree_height (h, Red)).
```

- We claim that this generator produces only RB trees.

```
QuickChick (insert_preserves_rb_checker genRBTree).

+++ OK, passed 10000 tests
```

Can we do better?

```
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))
                                    genNat (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'))
                             genNat (genRBTree_height (h', c')) end.

Definition genRBTree := bindGen genNat (fun h ⇒ genRBTree_height (h, Red)).
```

- We claim that this generator produces only RB trees.

```
QuickChick (insert_preserves_rb_checker genRBTree).

+++ OK, passed 10000 tests
```

- It seems that it works well in practice

Are we there yet?

- The previous generator is dubious
  (complex + unverified $\rightarrow$ dubious)

Are we there yet?

- The previous generator is dubious
  (complex + unverified $\rightarrow$ dubious)
- This generator also pass all the tests with no discards

```
Definition genRBTree := returnGen Leaf.
```

Our framework

- How do we know that we are generating all possible RB trees and only them?
- **Idea:** Assign semantics to each generator mapping them to the support of the underlying probability distribution

Our framework

- How do we know that we are generating all possible RB trees and only them?
- **Idea:** Assign semantics to each generator mapping them to the support of the underlying probability distribution
- How do we know that we are testing the logical proposition we started with?
- **Idea:** Assign semantics to each checker mapping it to the logical proposition that it tests.

```
semGen : ∀ A : Type, G A → set A


semChecker : Checker → Prop
```

Formal proof

```
Lemma semRBTree : semGen genRBTree ≡ [set t | is_redblack t].
```

Formal proof

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

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

Formal proof

```
Lemma semRBTree : semGen genRBTree ≡ [set t | is_redblack t].
```

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

```
Lemma insert_preserves_rb_checker_correct:
  semChecker (insert_preserves_rb_checker genRBTree)
  ↔ insert_preserves_rb.
```

Formal proof

```
Lemma semRBTree : semGen genRBTree ≡ [set t | is_redblack t].
```

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

```
Lemma insert_preserves_rb_checker_correct:
  semChecker (insert_preserves_rb_checker genRBTree)
  ↔ insert_preserves_rb.
```

- Complete example: 150 lines of proofs for 236 lines of definitions.

Semantics

- Generator type : `Definition G A = nat → RandomSeed → A.`

```
Definition semGenSize {A : Type} (g : G A) (size : nat) : set A :=
    ⋃_{seed ∈ Seeds} g size seed.

Definition semGen {A : Type} (g : G A) : set A :=
    ⋃_{size ∈ ℕ} semGenSize g size.
```

Semantics

- Generator type : `Definition G A = nat → RandomSeed → A.`

```
Definition semGenSize {A : Type} (g : G A) (size : nat) : set A :=
    ⋃_{seed ∈ Seeds} g size seed.

Definition semGen {A : Type} (g : G A) : set A :=
    ⋃_{size ∈ ℕ} semGenSize g size.
```

- Checkers are internally represented as generators of testing results

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

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

Size Abstraction

- Abstracting of sizes is not always possible! In the general case the semantics and the specifications need to be size parametric.

```
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.
```

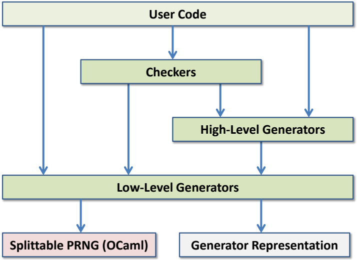
- Size abtraction only possible for *unsized* and *size-monotonic* generators

```
Lemma semBindSizeMonotonic :
  ∀ {A B} (g : G A) (f : A → G B)
         `{SizeMonotonic _ g} `{∀ a, SizeMonotonic (f a)},
    semGen (bindGen g f) ≡ \bigcup_(a in semGen g) semGen (f a).
```

- We provide size parametrized specifications for all of the combinators along with unsized specifications

Foundational Verification

- Using our possibilistic semantics we verify QuickChick all the way down relying on a very small set of assumptions
- We verify all the combinators of QuickChick providing a library of generic lemmas that can be used in a compositional way



Assumptions

- QuickChick's PRNG (pseudorandom number generator) is written in OCaml
- Low-level operations (such as random seed handling, generation of natural numbers, etc.) and their specifications are axiomatized in Coq
- We could remove most of the axioms by implementing PRNG in Coq
- One axiom would remain
  - The type of random seeds is infinite
- Our model abstracts away from mathematical randomness (probabilities), which is an idealization of pseudorandomness

Larger case study: Testing Noninterference

- We verified existing generators used in complex testing infrastructure for an information flow control (IFC) machine
- Generators used to produce pairs of indistinguishable states
- We proved that the generators were sound and complete w.r.t a subset of all possible indistinguishable states
- The process revealed bugs in generation
- Minimal changes to existing testing code were required
- 2000 lines of proofs for 2000 lines of Coq code (1000 lines of definitions and 1000 lines of generation code)

Conclusion and Future work

- Coq framework for verified PBT, integrated in QuickChick
  - `https://github.com/QuickChick`
- First verified QuickCheck implementation
- We avoid probabilisting reasoning at all level using possibilistic semantics
- Modular, scalable, minimal changes to existing code
- Future work: Reduce verification effort (typeclass automation, certificate producing testing automation)

Thank You!

Questions?

Related work

- Dybjer *et al.* first proposed the idea of verified generators (completeness property)
- Focaltest: Verified tool that automatically generates test data that satisfy MC/DC coverage for preconditions using constraint reasoning
- HOL-TestGen: Introduced *explicit test-hypotheses* that represent what remains to be proved

Examples of specifications

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

Lemma semBindUnsized1 :
  ∀ A B (g : G A) (f : A → G B) `{Unsized _ g},
    semGen (bindGen g f) ≡ \bigcup_(a in semGen g) semGen (f a).

Lemma semFmap :
  ∀ A B (f : A → B) (g : G A), semGen (fmap f g) ≡ f @: semGen g.

Lemma semOneOf : ∀ A (g0 : G A) (gs : list (G A)),
  semGen (oneOf (g0 ;; gs)) ≡ \bigcup_(g in (g0 :: gs)) semGen g.

Lemma semListOfUnsized:
  ∀ {A} (g : G A) (k : nat) `{ Unsized _ g},
  semGen (listOf g) ≡ [set l | l \subset semGen g ].
```