QuickChick: A Coq Framework For Verified Property Based Testing

Zoe Paraskevopoulou

September 8, 2014

Overview

- **Goal**: Trust high-level logical propositions instead of executable testing code
  - Gain confidence that the right conjecture is being tested
  - Gain confidence that the testing is thorough
- **Means**: Automatically map executable testing code (*checkers*) to logical propositions
  - Reasoning about probabilistic programs (*generators*)
    - map them to sets of outcomes
- **Evaluation**: Application to a number of sizable case studies
  - Modularity
  - Scalability
  - Minimal changes to existing code

Property-Based Testing

- Popularized by QuickCheck in the FP community
- Achieves a high level of automation
  - Randomly Generated Data
    - No need to maintain test suites
  - The programs are being tested against specifications
    - No need for human oracle
- The user has to write
  - Generators
    - Fine-tuning
    - Generation of used defined data types
  - Checkers
    - Programs that test the desired specification

QuickChick

- Randomized property-based testing framework for Coq
    - More precisely a port of QuickCheck in Coq
- Checkers need to be executable
    - No way to directly refute proof goals
- Written in Gallina
- Uses extraction to OCaml for:
    - Acquiring random seeds
    - Efficient execution
    - Generation of numeric (`nat` and `Z`) and boolean values

The value of counterexamples

- QuickChick returns counterexamples for falsifiable conjectures
  - Support for shrinking
    - try to isolate the part of the failing input that triggers the failure
- A counterexample could indicate:
  - errors in program
    - fix bug
  - errors in specifications
    - reformulate checker
- Valuable feedback to understand and fix errors

But . . .

How much confidence can we have about the program under test adhering its specifications, when the testing cannot find any more bugs?

Reasons for inadequate testing

Bugs in generators

- May fail to cover sufficiently the input space
  - Some counterexamples may never generated
- May fail to satisfy preconditions of conditional specifications
  - A big portion of the generated data can be discarded

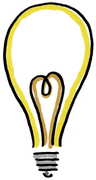
Bugs in checkers

- May fail to capture the desired specifications
  - Too strong preconditions, faulty definitions, . . .

VeriQuickChick

Idea

- Extend QuickChick so it automatically relates checkers to Coq propositions that capture the conjecture under test.
- Manually prove these propositions equivalent to the desired high-level declarative specifications



Guarantee

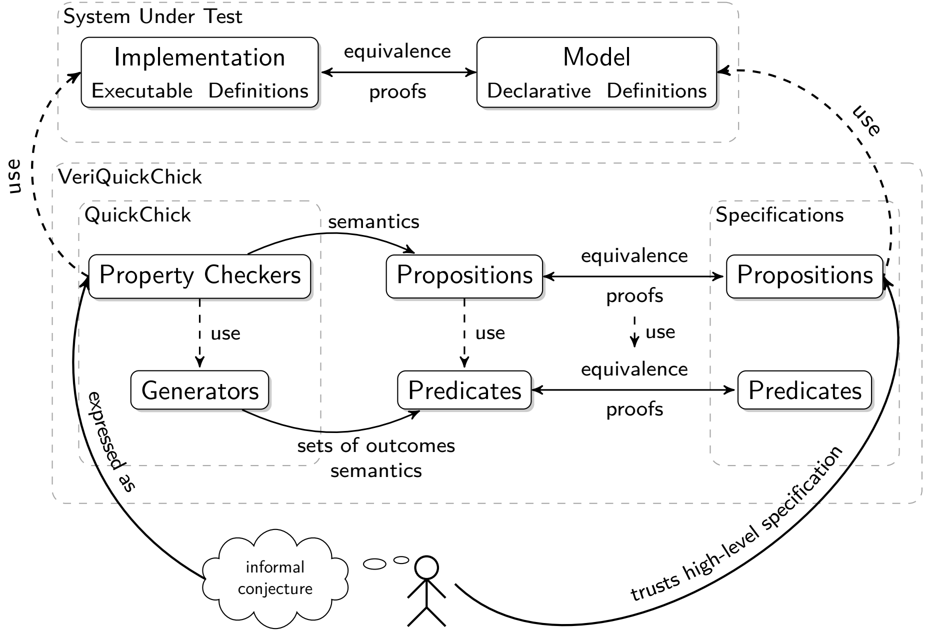
If we could enumerate the output space of the generators used to by the checker without producing any counter examples then we would have a proof by exhaustion for the desired high-level specification

Strategy I

- Map generators to *sets of outcomes*, i.e. the sets of values that have non-zero chance of being generated
  - Use logical predicates to represent sets
    - An element `a : A` belongs to a set that is represented by `P : A → Prop` if an only if `P a`
- Map checkers to logical propositions using the sets of outcomes of the generators that they use
  - If a generator `G` for type `A` is mapped to `S : A → Prop` then `forAll G f` is mapped to $\forall\ x,\ \mathtt{S}\ x \rightarrow \mathtt{f}\ x = \mathtt{true}$

Strategy II

- Use the sets of outcomes semantics of generators to prove:
    - *soundness*
        - All the values that are generated satisfy a certain predicate
    - *completeness*
        - All the values that satisfy a certain predicate can be generated
    - *correctness*
        - soundness + completeness
- Use the logical predicates obtained from checkers to prove that they correspond the desired declarative specification


System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
Specifications
semantics
Property Checkers
Propositions
equivalence
proofs
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

Generators

- Generators are written using a library of combinators (e.g. bind, return, elements, frequency, . . .)
  - *Primitive* combinators: they depend from the internal generator representation
  - *Non-primitive* combinators: they are built on top of other combinators
- *Overload* combinators with two kinds of semantics
  - actual generation semantics
  - sets of outcomes semantics
- Abstract from the generator representation: Make generators parametric in the generator type constructor
  - Instantiate them with the set representation to map them to sets of outcomes
  - or with the actual generator representation to generate data

Set Representation

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

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

Infix "⟷" := set_eq.
```

- Very compact set representation
  - Easily models infinite sets
  - Proof-oriented: facilitates reasoning for set membership
- Primitive combinators need to be implemented differently for each type constructor

Type classes to the rescue

```
Class GenMonad (M : Type → Type) :=
  {
    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 : Type} `{ Random A}, A * A → M A;
    sized : ∀ {A : Type}, (nat → M A) → M A;
    suchThatMaybe : ∀ {A : Type}, M A → (A → bool) →
                                  M (option A);
  }.
```

- All the primitive combinators are included in the type class
- Both generator type constructors are instances of this type class
- The set of outcome definitions of the primitive combinators is **axiomatic**
- The methods of the type class are implicitly parameterized by the type constructor and the corresponding instance

Axioms

$$\begin{aligned}
\texttt{returnGen}\ a &\equiv \{\ x \ \mid x = a\ \} \\
\texttt{bindGen}\ G\ f &\equiv \{\ x \ \mid \exists\ g,\ G\ g\ \wedge\ f\ g\ x\ \} \longleftrightarrow \bigcup_{g \in G} f\ g \\
\texttt{fmapGen}\ f\ G &\equiv \{\ x \ \mid \exists\ g,\ G\ g\ \wedge\ x = f\ g\} \\
\texttt{choose}\ (lo, hi) &\equiv \{\ x \ \mid lo \leq x \leq hi\ \} \\
\texttt{sized}\ f &\equiv \{\ x \ \mid \exists\ n,\ f\ n\ x\ \} \longleftrightarrow \bigcup_{n \in \mathbb{N}} f\ n \\
\texttt{suchThatMaybe}\ g\ P &\equiv \{\ x \ \mid x = None\ \vee \\
&\qquad \exists\ y,\ x = Some\ y\ \wedge\ g\ y\ \wedge\ P\ y\ \}
\end{aligned}$$

Non-primitive combinators

- Non-primitive combinators are built on top of the interface provided by the type class
- We move the definitions of non-primitive combinators inside a section in which we assume in context a type constructor which is instance of the `AbstractGen` type class.
    - No modification to their code is required
- Primitive combinators used are automatically instantiated with the type constructor and the instance assumed in context

Non-primitive combinators

Example

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

  Definition oneof {A : Type} (def: Gen A) (gs : list (Gen A))
  : Gen A :=
    bindGen (choose (0, length gs - 1)) (fun n ⇒
    nth def gs n).

    ...
    ...

End Utilities.
```

Lemma Library for Non-primitive Combinators

- Using the sets of outcomes semantics we prove correctness for each non-primitive generator combinator
- These lemmas can be used in proofs about user defined generators that use the combinators
  - less proof duplication and reusability
  - independence from the implementation of combinators
  - compositional and more robust proofs

Lemma Library for Non-primitive Combinators

Examples

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

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

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

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).


...
...
```

Checkers

- Checkers are essentially generators of testing results
- We map them to a proposition that holds iff all the results that belong to the sets of outcomes are successful
  - The result of a test input is successful if it is equal with the expected
- The simplest form of checkers are boolean predicates
- More complex checkers can be written by utilizing property combinators
  - change the expected outcome, change default generators, instrumentation
  - We provide a library of correctness lemmas

Checkers

- Checkers are represented internally with the type operator `Property`.

```
Definition Property (Gen : Type → Type) := Gen QProp.
```

- We use the function `semProperty` to map them to logical propositions

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

Testable type class

- Testable type class provides a canonical way of turning types that can be tested into a Property
- Anything testable can be mapped to a proposition

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

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

Lemma Library for Checker Combinators I

forAll and implication Lemmas

```
Lemma semForAll :
  ∀ {A prop : Type} {H1 : Testable prop} {H2 : Show A} (gen : Pred A)
    (f : A → prop),
    semProperty (forAll gen f) ↔ ∀ a : A, gen a → semTestable (f a).

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

Lemma Library for Checker Combinators II

Lemmas for specific testable types

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

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


...
...
```

`arbitrary` here is a generator for elements of type `A`. It

s a method of the `Arbitrary` type class that provides a

common interface for generation. `Testable` type class

use by default these generators to derive a `Property`

Lemma Library for Checker Combinators III

Identity Lemmas

Some combinators do not affect the testing outcome. They are used for instrumentation purposes.

```
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.
```

Case Study

Red-black Trees

Red-Black Trees

- A self-balancing data structure
- Binary trees with an additional color label to each node

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

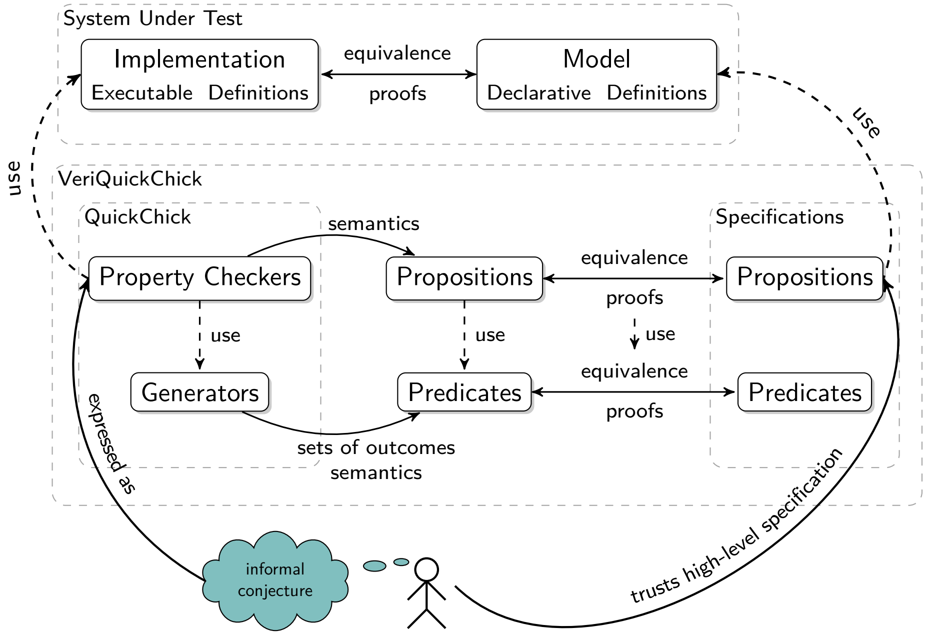
- The should preserve the following invariants
  - The root is always black
  - The leaves are empty and black
  - For each node the path to each possible leaf has the same number of black nodes
  - Red nodes can only have black children
- If the invariants are preserved then the longest path from the root is at most two times longer that the shortest
- The `insert` method should preserve the invariant
  - We want to test that!

Red-Black Trees

- A self-balancing data structure
- Binary trees with an additional color label to each node

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

- The should preserve the following invariants
  - 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
- If the invariants are preserved then the longest path from the root is at most two times longer that the shortest
- The `insert` method should preserve the invariant
  - We want to test that!

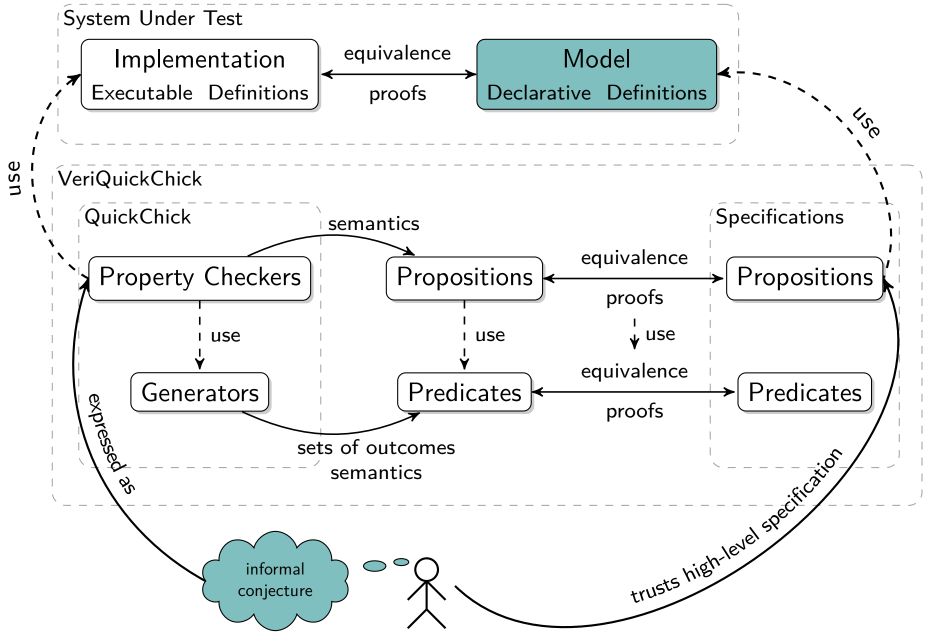

System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Specifications
Property Checkers
Propositions
equivalence
proofs
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

The 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).
```

- `is_redblack t c h` means that `t` is a subtree of a well-formed RB tree
  - in *color-context* `c` (the color of the parent node)
  - with *black-height* `t` (# black nodes in each path to a leaf)
- A tree t satisfies the RB invariant iff:

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



System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Specifications
Property Checkers
Propositions
equivalence
proofs
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

The Red-Black Invariant

- `insert` should preserve the invariant:

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

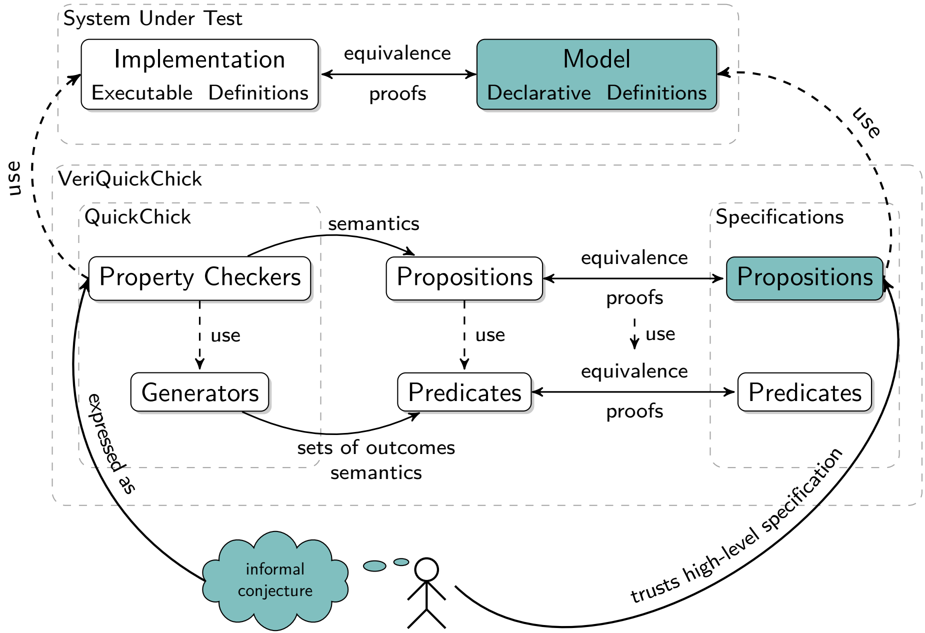
- In order to be able to test this we need
  - A decision procedure to determine whether a tree satisfies the RB invariant
  - A generator for RB trees
    - Should generate only trees that satisfy the invariant
    - Filtering out ill-formed RB trees is not an option

The Red-Black Invariant

- `insert` should preserve the invariant:

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

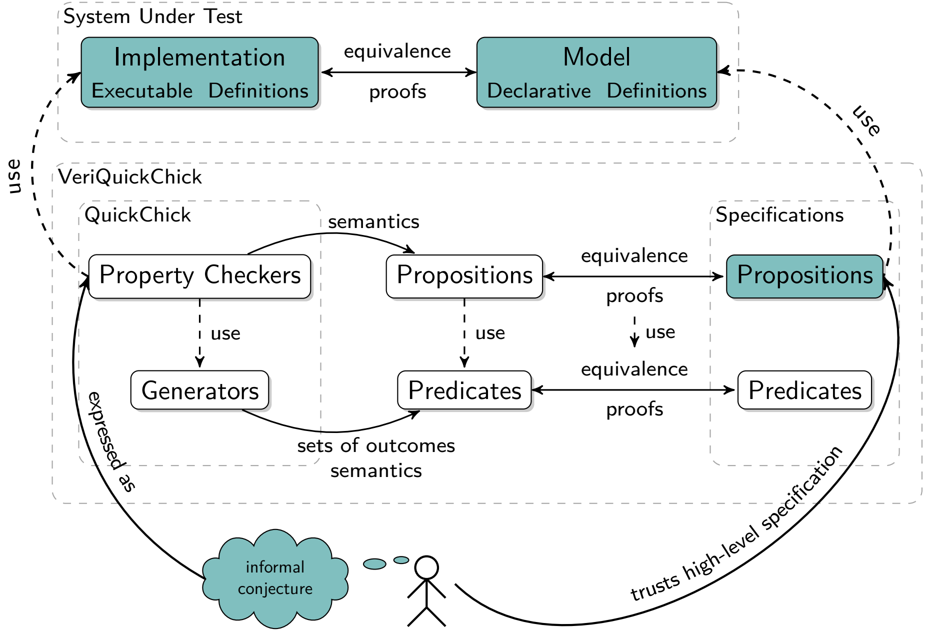
- In order to be able to test this we need
  - A decision procedure to determine whether a tree satisfies the RB invariant
  - A generator for RB trees
    - Should generate only trees that satisfy the invariant
    - Filtering out ill-formed RB trees is not an option


System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Specifications
Property Checkers
Propositions
equivalence
proofs
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
expressed as
sets of outcomes
semantics
informal
conjecture
trusts high-level specification

Executable Definitions

We need a decisional procedure to determine whether a tree satisfies the RB invariant.

```
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.
```

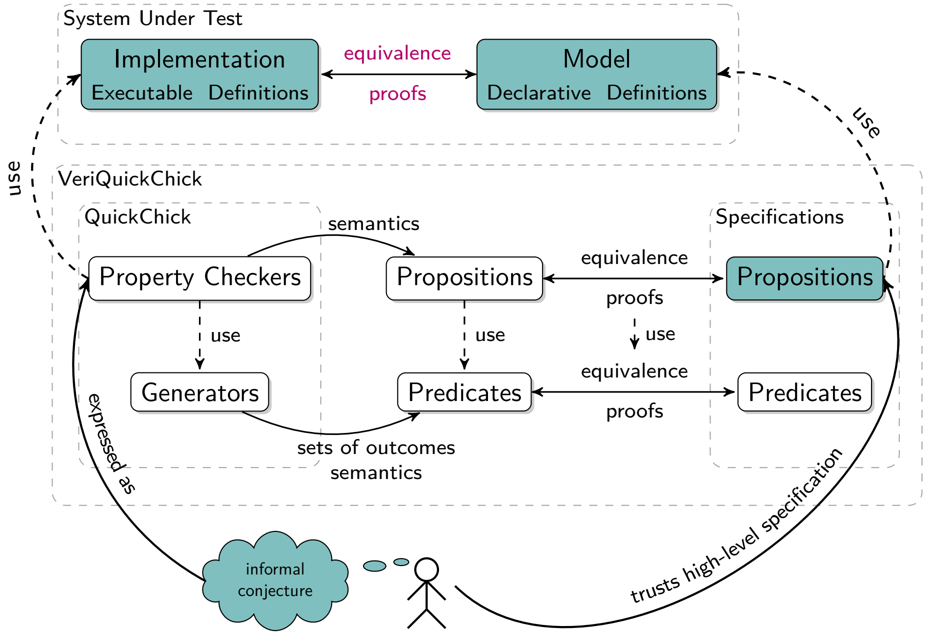


System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Specifications
Property Checkers
Propositions
equivalence
proofs
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

Executable Definitions

Does it correspond to the inductive definition? Yes!

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


System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Property Checkers
Propositions
equivalence
proofs
Specifications
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

Red-black Tree Generator

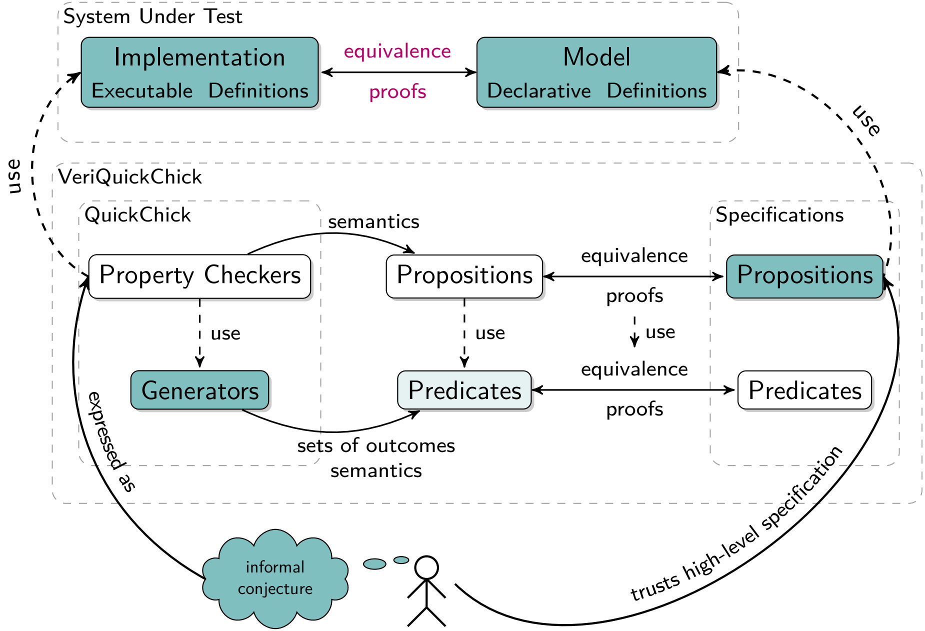
```
Section Generators.
  Context {Gen : Type → Type}
          {H: GenMonad Gen}.

  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 ⇒ ...
    end.

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

End Generators.
```



System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Property Checkers
Propositions
equivalence
proofs
Specifications
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
informal
conjecture
trusts high-level specification

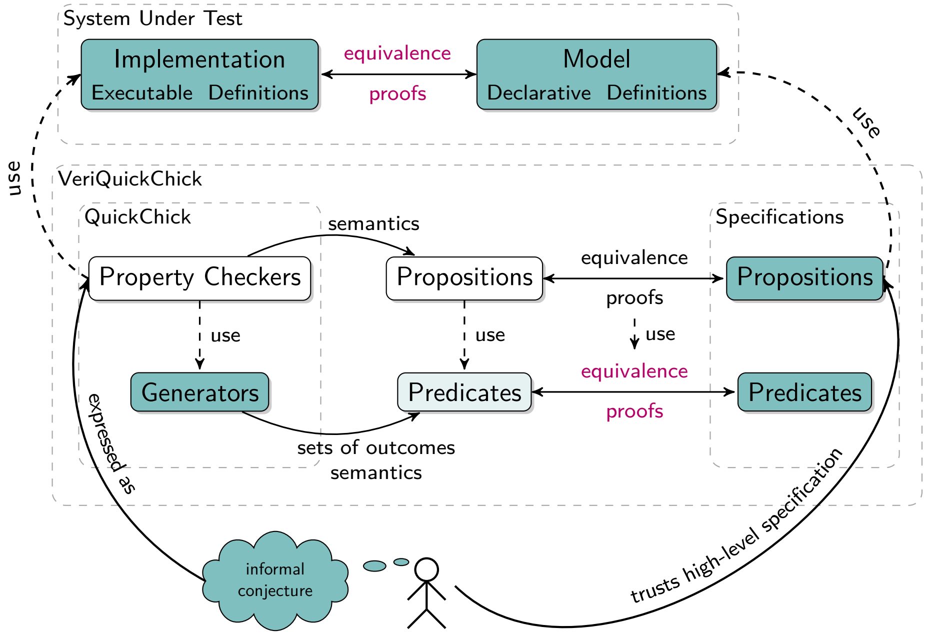
Correctness for Red-black Tree Generator

We want to prove that the generator generates only trees that satisfy the RB invariant and also that it can generate all the possible trees that satisfy the RB invariant.

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

We need an intermediate lemma

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


System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
Property Checkers
semantics
Propositions
equivalence
proofs
Specifications
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
informal
conjecture
trusts high-level specification

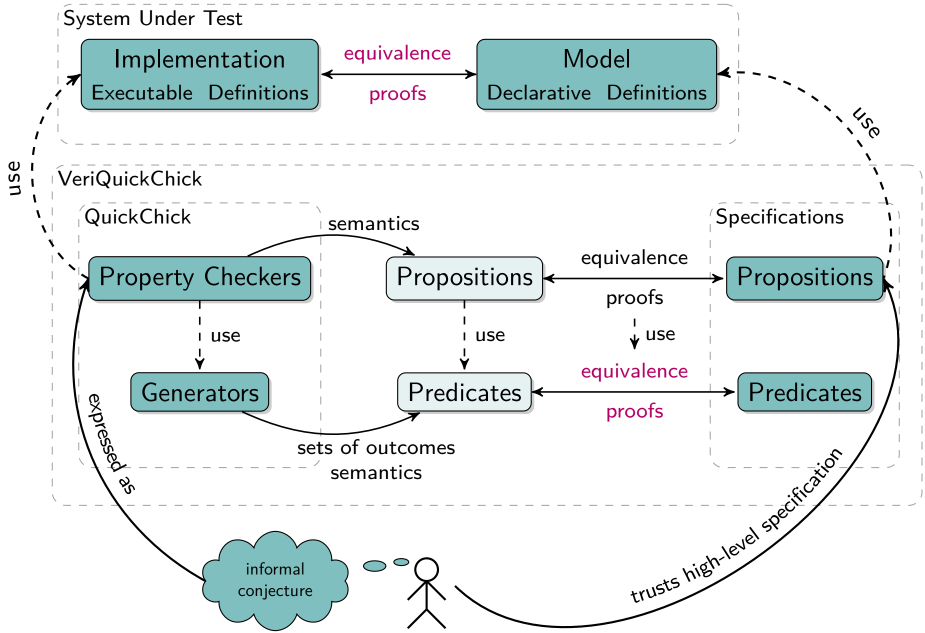
Checker

We can now write the checker for the property:

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

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

End Checker.
```

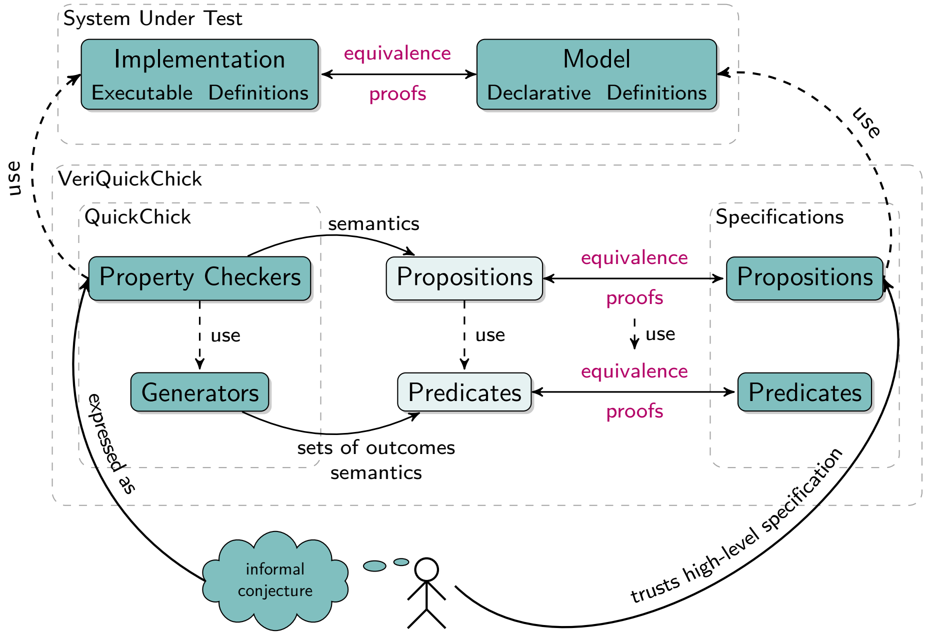

System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Property Checkers
Propositions
equivalence
proofs
Specifications
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

Correctness for Checker

.. and prove that it indeed tests the right thing

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

Lemma insert_is_redblack_checker_correct:
  semProperty insert_is_redblack_checker ↔ insert_is_redblack.
```



System Under Test
Implementation
Executable Definitions
equivalence
proofs
Model
Declarative Definitions
use
use
VeriQuickChick
QuickChick
semantics
Specifications
Property Checkers
Propositions
equivalence
proofs
Propositions
use
use
use
Generators
Predicates
equivalence
proofs
Predicates
sets of outcomes
semantics
expressed as
trusts high-level specification
informal
conjecture

Conclusion

- We provide a mechanism to verify that a conjecture under test conforms to a high-level specification
- We facilitate reasoning for probabilistic programs
  - set of outcomes abstraction
- We proved high-level specifications about combinators
  - Proofs: 600 LOC
  - First step towards a fully verified PB testing framework
- We applied our methods to verify complex generators used to test an IFC machine
  - Generators: 350 LOC / Proofs: 900 LOC
  - ability to verify existing code
  - scalability
  - modularity
- Although reduced, manual effort is still required for the proofs

Future Work

- Remove the axioms by proving that the sets of outcomes of primitive combinators are indeed the those we assume
  - Fully verified PB testing framework
  - This would require deeper integration in Coq
    - Reasoning about random seed and generators of numeric and boolean values
- Verification of a framework for synthesizing generators from specifications
  - automation + formal guarantees
- Facilitate reasoning for the underlying probability distibutions
  - instantiate generators with probability monad

Thank You!

Questions?

Checkers (internals) I

```
Record Result :=
  MkResult {
      ok : option bool;  (* Testing outcome *)
      expect : bool;     (* Expected outcome *)
      ...                (* Other fields used for tracing *)
    }.

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

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

Definition Property (Gen : Type → Type) := Gen QProp.
```

Checkers (internals) II

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

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

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