# A Coq Framework For Verified Property-Based Testing

(part of QuickChick)

Cătălin Hrițcu

INRIA Paris-Rocquencourt

(Prosecco team, Place d'Italie office)

# Problem: proving in Coq is very costly

- **My proofs are boring, but designing security mechanisms is not**
  - definitions and properties often broken, and evolve over time

# Problem: proving in Coq is very costly

- **My proofs are boring, but designing security mechanisms is not**
  - definitions and properties often broken, and evolve over time
- **Proving does aid design ... but only at a very high cost**
  - most enlightenment comes from failed, not from successful proofs
    a failed proof attempt is a very costly way to discover a design flaw
    fixing flaws not always easy, might require serious redesign
  - failed proof attempt will generally not convince an engineer
    proving while designing is frustrating, tedious, time consuming

# Problem: proving in Coq is very costly

- **My proofs are boring, but designing security mechanisms is not**
  - definitions and properties often broken, and evolve over time
- **Proving does aid design ... but only at a very high cost**
  - most enlightenment comes from failed, not from successful proofs

    a failed proof attempt is a very costly way to discover a design flaw

    fixing flaws not always easy, might require serious redesign
  - failed proof attempt will generally not convince an engineer

    proving while designing is frustrating, tedious, time consuming

**Even when design correct & stable, proving still costly**

  - countless iterations for discovering lemmas and invariants
  - my proofs are often “fragile”, so the cost of each iteration is high

# Problem: proving in Coq is very costly

- **My proofs are boring, but designing security mechanisms is not**
  - definitions and properties often broken, and evolve over time
- **Proving does aid design … but only at a very high cost**
  - most enlightenment comes from failed, not from successful proofs

- This is the itch I'm trying to scratch
  - many people seem to have similar itches though

- **Even when design correct & stable, proving still costly**
  - countless iterations for discovering lemmas and invariants
  - my proofs are often "fragile", so the cost of each iteration is high

# Could **testing** help with this problem?

- Can property-based testing
  - lower the cost of formal proofs?
  - become an important part of the theorem proving process in Coq?

# Could **testing** help with this problem?

- Can property-based testing
  - lower the cost of formal proofs?
  - become an important part of the theorem proving process in Coq?
- Yes, I believe / hope so

  own recent positive experience with testing

  I'm not the only one (e.g. Isabelle, FocalTest, ...)

# Could **testing** help with this problem?

- Can property-based testing
  - lower the cost of formal proofs?
  - become an important part of the theorem proving process in Coq?
- Yes, I believe / hope so

  own recent positive experience with testing

  I’m not the only one (e.g. Isabelle, FocalTest, ...)


We are basically just starting on this

- A lot of research & engineering work left

# Collaborators

**Arthur Azevedo de Amorim**
(UPenn, recent Inria intern)

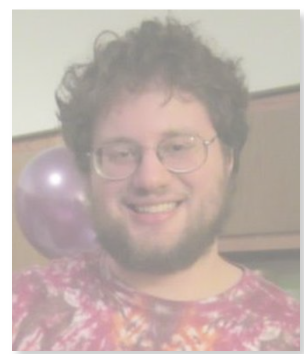
**Maxime Dénès**
(Inria)

**John Hughes**
(Chalmers)

**Leo Lampropoulos**
(UPenn)

**Zoe Paraskevopoulou**
(ENS Cachan, MPRI,
recent Inria intern)

**Benjamin Pierce**
(UPenn)



**Antal Spector-Zabusky**
(UPenn)

**Dimitris Vytiniotis**
(MSR Cambridge)

# This talk

- Property-based testing with QuickChick
  - Our QuickCheck clone for Coq (prototype plugin)
  - Everything at [https://github.com/QuickChick](https://github.com/QuickChick)
- Framework for verified property-based testing

- Other things we are doing that I won’t discuss today
  - Case studies: noninterference, security monitors, type-checkers
  - Relating executable and declarative artifacts in Coq/SSReflect
  - Language for property-based generators
  - Evaluating testing quality: polarized mutation testing

**Zoe Paraskevopoulou**
(ENS Cachan, MPRI,
recent Inria intern)

**Maxime Dénès**
(Inria)

**Leo Lampropoulos**
(UPenn)

Property-based testing with QuickChick

# TESTING RED-BLACK TREES

# Red-Black Tree Implementation

```
Inductive color := Red | Black.

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

Definition balance rb t1 k t2 :=
  match rb with
    | Red => Node Red t1 k t2
    | _ =>
      match t1 with
        | Node Red (Node Red a x b) y c =>
          Node Red (Node Black a x b) y (Node Black c k t2)
        | Node Red a x (Node Red b y c) =>
          Node Red (Node Black a x b) y (Node Black c k t2)
        | a => match t2 with
                 | Node Red (Node Red b y c) z d =>
                   Node Red (Node Black t1 k b) y (Node Black c z d)
                 | Node Red b y (Node Red c z d) =>
                   Node Red (Node Black t1 k b) y (Node Black c z d)
                 | _ => Node Black t1 k t2
               end
      end
  end.
```

# Red-Black Trees Implementation

```
Inductive color := Red | Black.

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

Fixpoint ins x s :=
  match s with
    | Leaf => Node Red Leaf x Leaf
    | Node c a y b => if x < y then balance c (ins x a) y b
                      else if y < x then balance c a y (ins x b)
                           else Node c a x b
  end.

Definition makeBlack t :=
  match t with
    | Leaf => Leaf
    | Node _ a x b => Node Black a x b
  end.

Definition insert x s := makeBlack (ins x s).
```

# Declarative Proposition

```
(* Red-Black Tree invariant: declarative definition *)
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 := exists h, is_redblack' t Red h.

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

(* Declarative Proposition *)
Lemma insert_preserves_redblack_correct : insert_preserves_redblack.
Abort. (* if this wasn't about testing, we would just prove this *)
```

# Executable Definitions

```
(* Red-Black Tree invariant: executable definition *)

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

Definition is_black_balanced (t : tree) : bool :=
  isSome (black_height_bool t).
```

# Property Checker

```
Fixpoint has_no_red_red (t : tree) : bool :=
  match t with
  | Leaf => true
  | Node Red (Node Red _ _ _) _ _ => false
  | Node Red _ _  (Node Red _ _ _) => false
  | Node _ tl _ tr => has_no_red_red tl && has_no_red_red tr
  end.

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

```
Definition insert_is_redblack_checker : Gen QProp :=
  forAll arbitrary (fun n =>
  (forAll genTree (fun t =>
    (is_redblack_bool t ==>
     is_redblack_bool (insert n t)) : Gen QProp)) : Gen QProp).
```

# Custom Generator for Trees

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

Fixpoint genAnyTree_max_height (h : nat) : Gen tree :=
  match h with
  | 0 => returnGen Leaf
  | S h' =>
      bindGen genColor (fun c =>
      bindGen (genAnyTree_max_height h') (fun t1 =>
      bindGen (genAnyTree_max_height h') (fun t2 =>
      bindGen arbitraryNat (fun n =>
      returnGen (Node c t1 n t2)))))
  end.

Definition genAnyTree : Gen tree := sized genAnyTree_max_height.
```

# Running QuickChick

```
Extract Constant defSize => "5".
Extract Constant Test.defNumTests => "100".
QuickCheck testInsertNaive.
Extract Constant Test.defNumTests => "10000".
```

```
Warning: The extraction is currently set to bypass opacity,
the following opaque constant bodies have been accessed :
  eqnP idP iffP.
```

```
*** Gave up! Passed only 3 tests
Discarded: 200
```

# Finding a Bug

```
Fixpoint has_no_red_red (t : tree) : bool :=
  match t with
  | Leaf => true
  | Node Red (Node Red _ _ _) _ _ => false
  | Node Red _ _ (Node Red _ _ _) => false
  | Node _ tl _ tr => has_no_red_red tr && has_no_red_red tr
  end.
```

```
Extract Constant defSize => "5".
Extract Constant Test.defNumTests => "10000".
QuickCheck testInsertNaive.
```

```
Node Black (Node Red (Node Red (Leaf) 63 (Leaf)) 155 (Node Red (Leaf) 55 (Noc
*** Failed! After 4021 tests and 0 shrinks
```

# Property-Based Generator

```
Fixpoint genRBTree_height (h : nat) (c : color) :=
  match h with
    | 0 =>
      match c with
        | Red => returnGen Leaf
        | Black => oneof (returnGen Leaf)
                         [returnGen Leaf;
                           bindGen arbitraryNat (fun n =>
                           returnGen (Node Red Leaf n Leaf))]
      end
    | S h =>
      match c with
        | Red =>
          bindGen (genRBTree_height h Black) (fun t1 =>
          bindGen (genRBTree_height h Black) (fun t2 =>
          bindGen arbitraryNat (fun n =>
          returnGen (Node Black t1 n t2))))
        | Black =>  ..........

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

# Property-Based Generator at Work

```
Variable genTree : Gen tree.

Definition insert_is_redblack_checker : Gen QProp :=
  forAll arbitraryNat (fun n =>
  (forAll genTree (fun t =>
    (is_redblack_bool t ==>
     is_redblack_bool (insert n t)) : Gen QProp)) : Gen QProp).
```

```
Definition testInsert :=
  showDiscards (quickCheck (insert_is_redblack_checker genRBTree)).

Extract Constant defSize => "10".
Extract Constant Test.defNumTests => "10000".
QuickCheck testInsert.
```

```
Success: number of successes 10000
         number of discards 0
```

in less than 4 seconds

**Zoe Paraskevopoulou**
(ENS Cachan, MPRI,
recent Inria intern)

Are we testing the right property?

# VERIFIED PROPERTY-BASED TESTING

# Testing Code Can Be Wrong

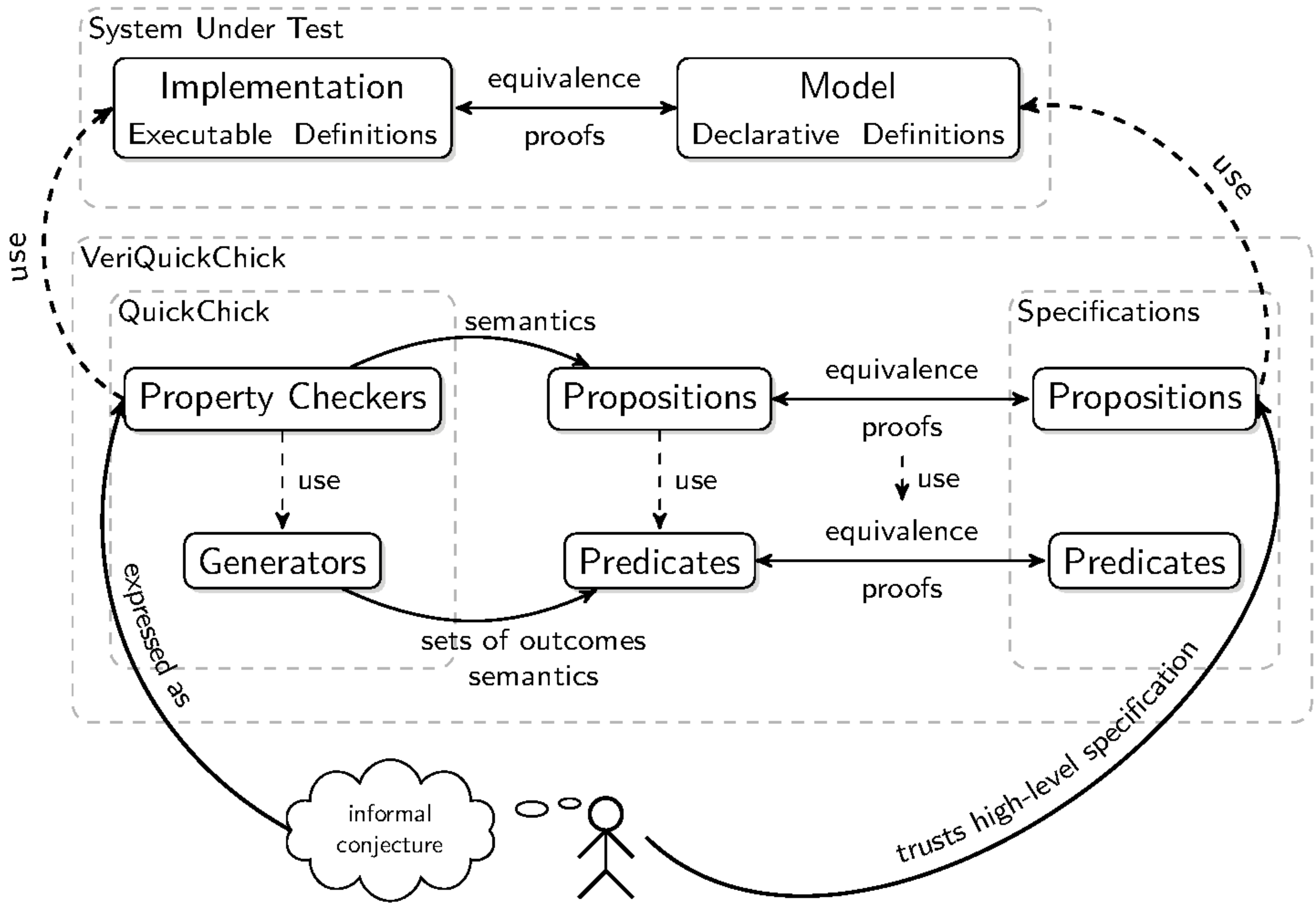
- QuickChick user has to write effective checkers and generators by hand
  - [working on a new language in which one can write both generator and checker as a single program]
  - errors can result in testing the wrong conjecture
  - randomness makes finding and fixing errors hard

# Testing Code Can Be Wrong

- QuickChick user has to write effective checkers and generators by hand
  - [working on a new language in which one can write both generator and checker as a single program]
  - errors can result in testing the wrong conjecture
  - randomness makes finding and fixing errors hard
- User generators and checkers
  + most of QuickChick itself written in Coq
  - **Can formally we verify them?**

# Verified Property-Based Testing

- Verification framework on top of QuickChick
- Prove correctness of generators and checkers with respect to their declarative specs
- **Main novelty: set of outcomes abstraction**
  - sem. of generator (Gen A) is an Ensemble (A -> Prop)
    - the set of values that can be generated with >0 probability
  - semantics of checker is a Coq proposition (Prop)
    - internally checkers are also generators (Gen Result)
    - all results are successful


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
informal
conjecture
trusts high-level specification

# Proving correctness of generators

```
Definition set_eq {A} (m1 m2 : Pred A) := forall A, m1 A <-> m2 A.
Infix "<-->" := set_eq (at level 70, no associativity) : pred_scope.
```

# Proving correctness of generators

```
Definition set_eq {A} (m1 m2 : Pred A) := forall A, m1 A <-> m2 A.
Infix "<-->" := set_eq (at level 70, no associativity) : pred_scope.

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

# Proving correctness of generators

```
Definition set_eq {A} (m1 m2 : Pred A) := forall A, m1 A <-> m2 A.
Infix "<-->" := set_eq (at level 70, no associativity) : pred_scope.

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

```
Lemma genColor_correct:
  genColor <--> all.
Proof.
  rewrite /genColor. intros c. rewrite elements_equiv.
  split => // _. left.
  destruct c;  by [ constructor | constructor(constructor)].
Qed.
```

# Proving correctness of generators

```
Definition set_eq {A} (m1 m2 : Pred A) := forall A, m1 A <-> m2 A.
Infix "<-->" := set_eq (at level 70, no associativity) : pred_scope.

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

Lemma elements_equiv :
  forall {A} (l: list A) (def : A),
    (elements def l) <--> (fun e => In e l \/ (l = nil /\ e = def)).

Lemma genColor_correct:
  genColor <--> all.
Proof.
  rewrite /genColor. intros c. rewrite elements_equiv.
  split => // _. left.
  destruct c;  by [ constructor | constructor(constructor)].
Qed.
```

# Proving correctness of generators

```
Definition set_eq {A} (m1 m2 : Pred A) := forall A, m1 A <-> m2 A.
Infix "<-->" := set_eq (at level 70, no associativity) : pred_scope.

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

Lemma elements_equiv :
  forall {A} (l: list A) (def : A),
    (elements def l) <--> (fun e => In e l \/ (l = nil /\ e = def)).

Lemma genColor_correct:
  genColor <--> all.
Proof.
  rewrite /genColor. intros c. rewrite elements_equiv.
  split => // _. left.
  destruct c;  by [ constructor | constructor(constructor)].
Qed.

Lemma genRBTree_height_correct:
  forall c h,
    (genRBTree_height h c) <--> (fun t => is_redblack' t c h).
```

# Proving correctness of generators

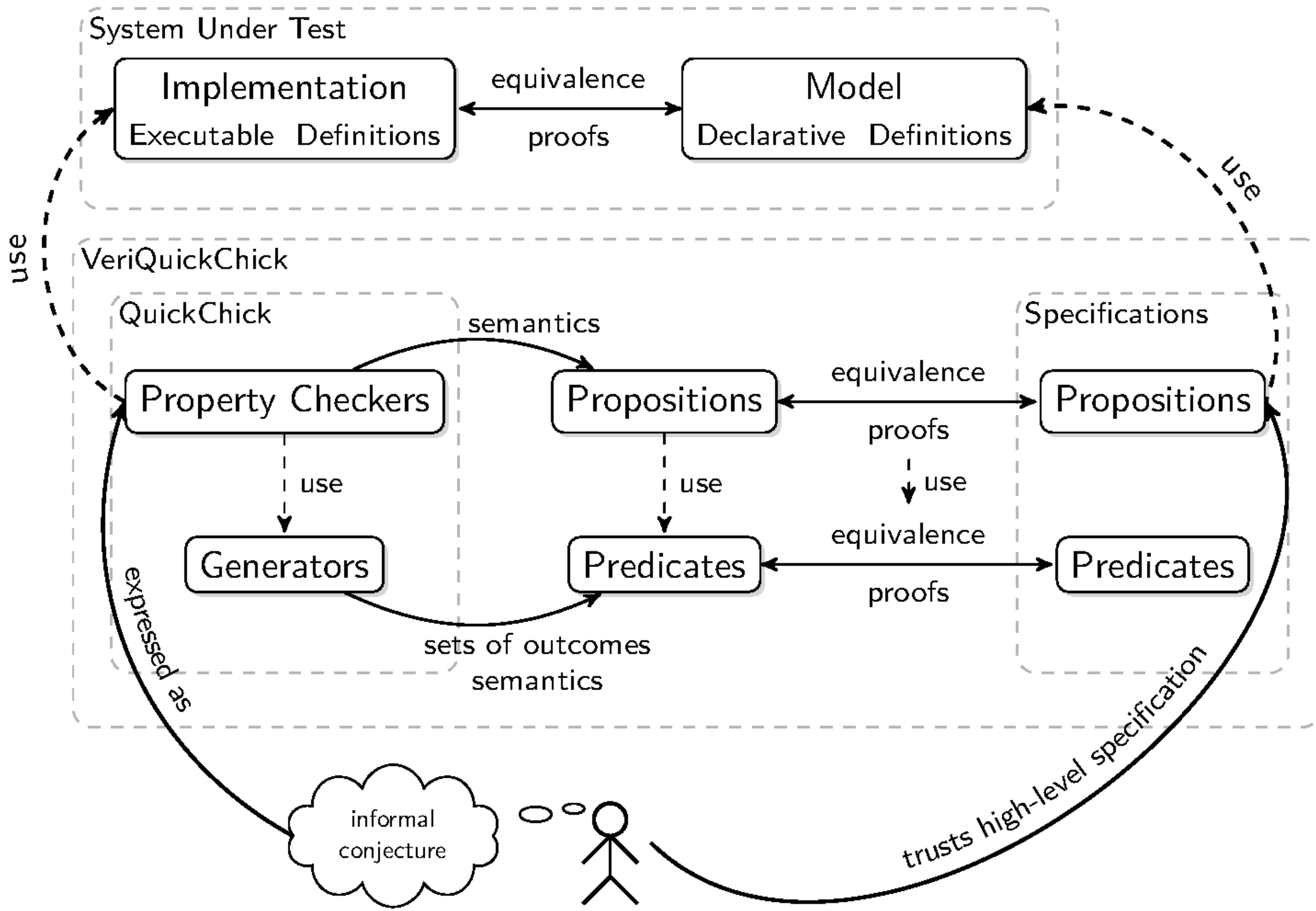
```
Definition set_eq {A} (m1 m2 : Pred A) := forall A, m1 A <-> m2 A.
Infix "<-->" := set_eq (at level 70, no associativity) : pred_scope.

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

Lemma elements_equiv :
  forall {A} (l: list A) (def : A),
    (elements def l) <--> (fun e => In e l \/ (l = nil /\ e = def)).

Lemma genColor_correct:
  genColor <--> all.
Proof.
  rewrite /genColor. intros c. rewrite elements_equiv.
  split => // _. left.
  destruct c;  by [ constructor | constructor(constructor)].
Qed.

Lemma genRBTree_height_correct:
  forall c h,
    (genRBTree_height h c) <--> (fun t => is_redblack' t c h).

Lemma genRBTree_correct:
  genRBTree <--> is_redblack.
```

# Proving correctness of checkers

Relating Executable and Declarative Definitions (SSReflect Style)

```
Lemma is_redblackP :
  forall (t : tree),
    reflect (is_redblack t) (is_redblack_bool t).
```

```
Lemma insert_is_redblack_checker_correct:
  semChecker (insert_is_redblack_checker genRBTree) <-> insert_preserves_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
informal
conjecture
trusts high-level specification

# Axioms for Primitive Combinators

$$\texttt{returnGen}\ a \equiv \{\ x \ \mid x = a\ \}$$

$$\texttt{bindGen}\ G\ f \equiv \{\ x \ \mid \exists\ g,\ G\ g\ \wedge\ f\ g\ x\ \} \longleftrightarrow \bigcup_{g \in G} f\ g$$

$$\texttt{fmapGen}\ f\ G \equiv \{\ x \ \mid \exists\ g,\ G\ g\ \wedge\ x = f\ g\}$$

$$\texttt{choose}\ (lo, hi) \equiv \{\ x \ \mid lo \leq x \leq hi\ \}$$

$$\texttt{sized}\ f \equiv \{\ x \ \mid \exists\ n,\ f\ n\ x\ \} \longleftrightarrow \bigcup_{n \in \mathbb{N}} f\ n$$

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

# Lemmas for Derived Generators

```
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).
```

# Lemmas for Checkers

```
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.
```

# Future Work

- More proof automation and infrastructure
  - changing to efficient data representations
  - SMT-based verif. for set of outcome abstraction?
- The first verified QuickCheck implementation
  - reduce the number of axioms
  - probabilistic verification?
- Verify property-based generator language
  - in general, manually verify reusable infrastructure
- *Motto: premature automation is the root of all evil*

# THANK YOU

Code at https://github.com/QuickChick