A Coq Framework For Verified Property-Based Testing

Cătălin Hrițcu

INRIA Paris

(part of QuickChick)

INVENTORS FOR THE DIGITAL WORLD

Coq Verification Is **Expensive**

- When designing and verifying real systems, ***most enlightenment comes from counterexamples***
- *but **finding counterexamples via failed proofs very costly***
- Want to ***find counterexamples as early as possible***
- ***Counterexample convinces engineer better than failed proof***
- ***Designs evolve***, definitions and properties often wrong
- Even when design correct & stable, proving still costly: countless iterations for ***discovering lemmas and invariants***
- ***this is the itch we're trying to scratch with QuickChick***

QuickChick: Property-Based Testing for Coq

- We believe that property-based testing can
  - lower the cost of Coq proofs
  - become a part of the Coq proving process (similarly to Isabelle, ACL2, PVS, TLA+, etc)
- Not there yet ... but at the moment we have
  - a working clone of Haskell’s QuickCheck
    - Prototype Coq plugin written mostly in Coq itself https://github.com/QuickChick
  - various other prototypes and experiments
  - lots of ideas we’re trying out

Collaborators


Arthur Azevedo de Amorim
(UPenn, recent Inria intern)


Maxime Dénès
(Inria)

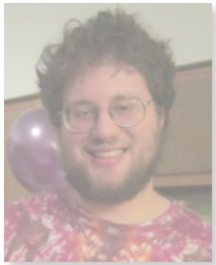

John Hughes
(Chalmers)


Cătălin Hrițcu
(Inria)


Leo Lampropoulos
(UPenn)


Zoe Paraskevopoulou
(ENS Cachan,
Inria intern
last summer)


Benjamin Pierce
(UPenn)


Antal Spector-Zabusky
(UPenn)


Dimitris Vytiniotis
(MSR Cambridge)


Li-yao Xia
(ENS Paris,
upcoming
Inria intern)

- writing our testing framework in Coq enables proving formal statements about testing itself
- this is the main topic of this talk

Verified Property-Based Testing? **Why?**

1. QuickChick is not push button
   - users will always have to write some code
     - **property checkers** (efficiently executable variants of properties)
     - **property-based generators** (producing data satisfying properties)
   - writing correct **probabilistic programs** is hard
   - easy to test things badly and not notice it until proving (e.g. test weaker property); this reduces benefit of testing
   - when testing finds no bugs, how can we know that we are testing things right? are we even testing the right thing?
     - **answer #1: formal verification**
     - answer #2: polarized mutation testing

Verified Property-Based Testing? **Why?**

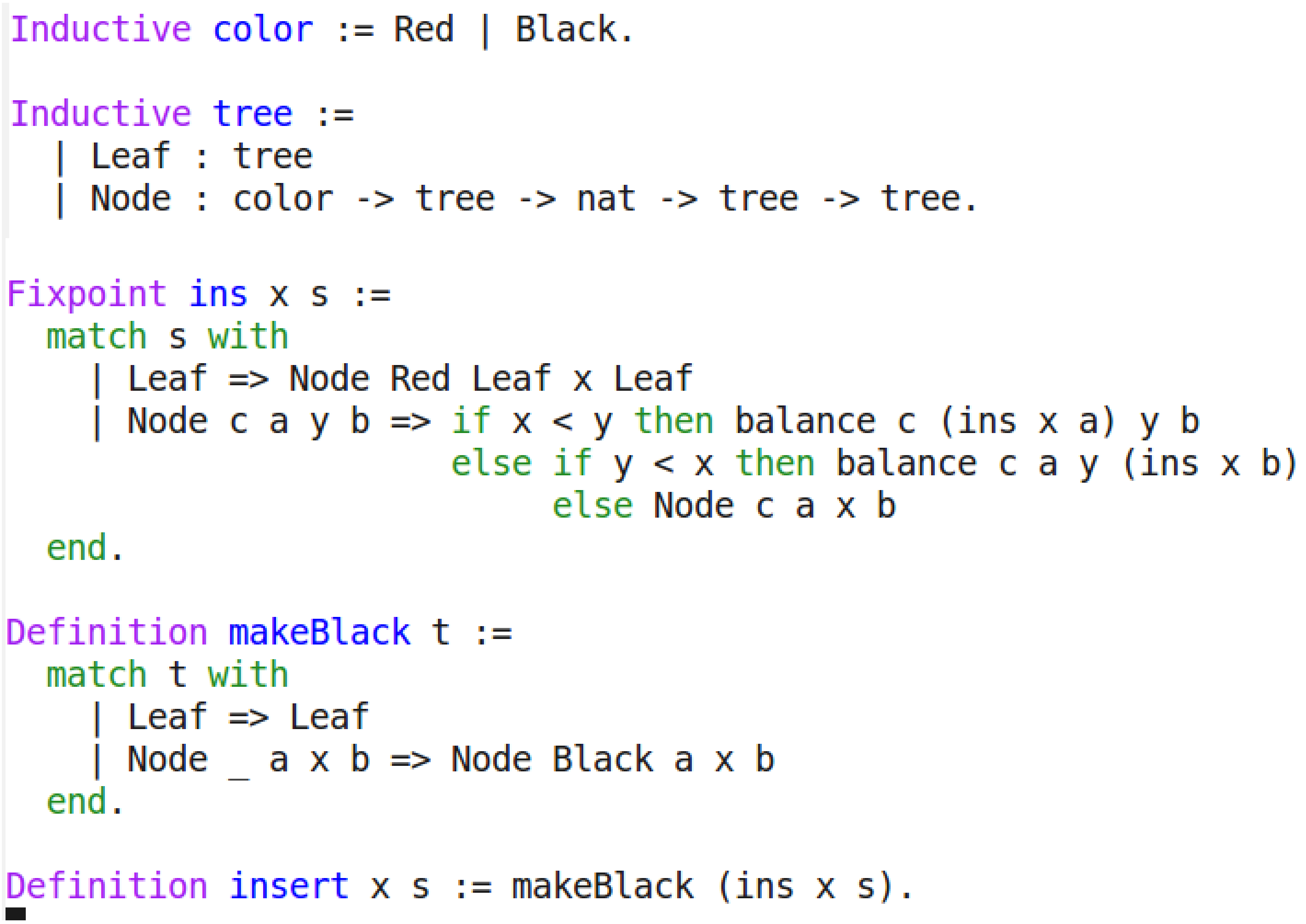
2. Need to trust QuickChick itself
   - Subtle bugs found in Haskell QuickCheck even after 14 years of widespread usage
   - The more smarts we add to QuickChick, the bigger this issue becomes
   - Any extension we make needs to be correct
     - e.g. we would like to work out the metatheory of our upcoming property-based generator language
     - but for this we need at first *define* what generator and checker correctness means

A Coq Framework for Verified PBT

- Formally verify QuickChick generators and checkers
  - wrt high-level properties they are supposed to test
- Methodology for verification of probabilistic programs
  - **abstraction**: reasoning about the **sets of outcomes** a they can produce with non-zero probability
- Framework integrated in QuickChick, used to verify
  - almost all the QuickChick combinators
  - red-black trees and noninterference examples
- Modular, scalable, requires minimal code changes

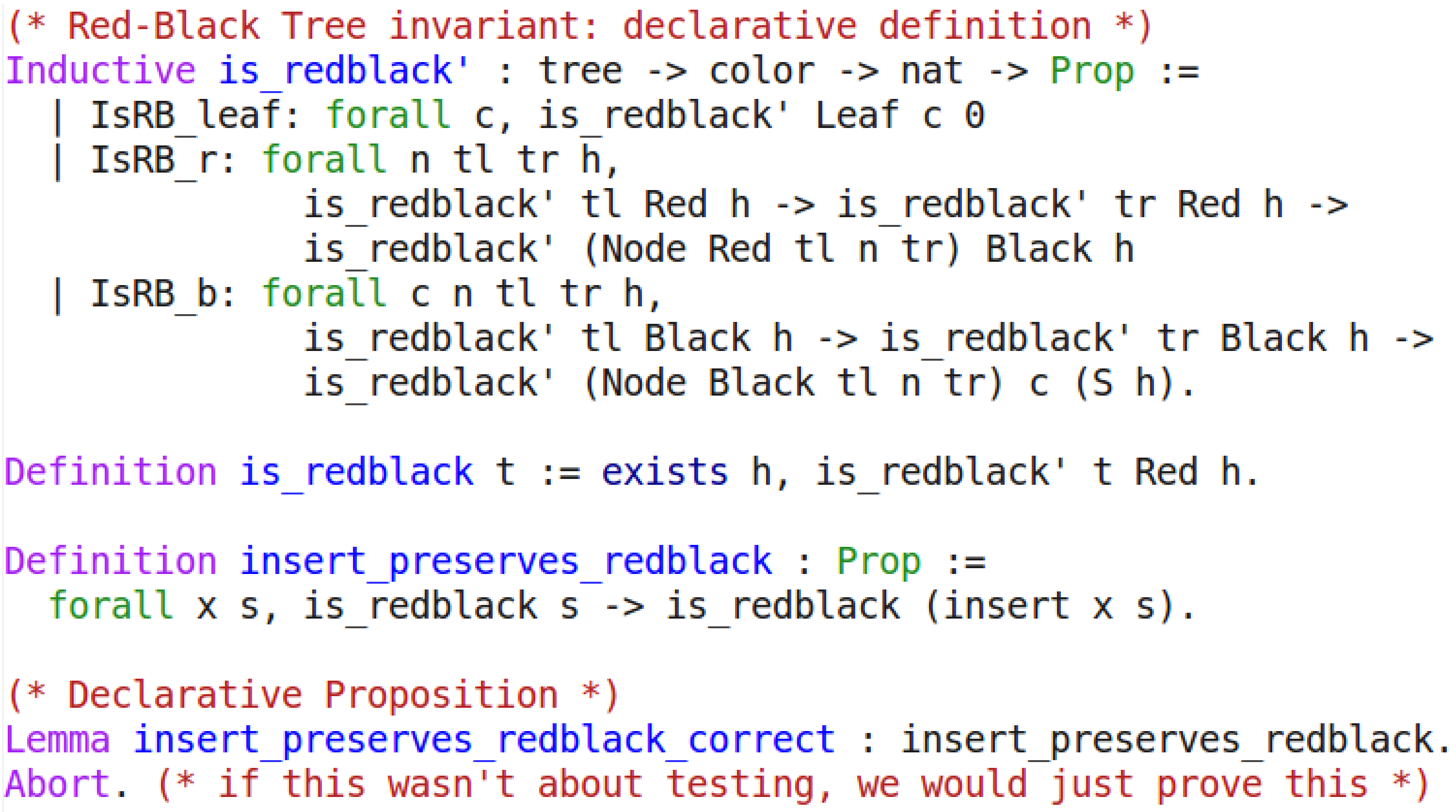
A QUICK INTRODUCTION TO QUICKCHICK

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).
```

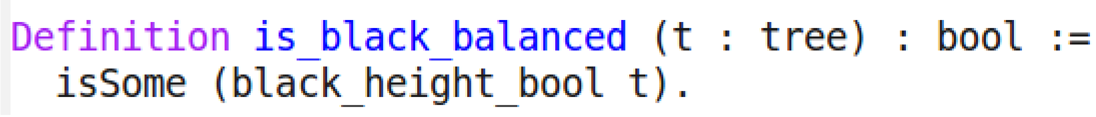
Declarative Proposition



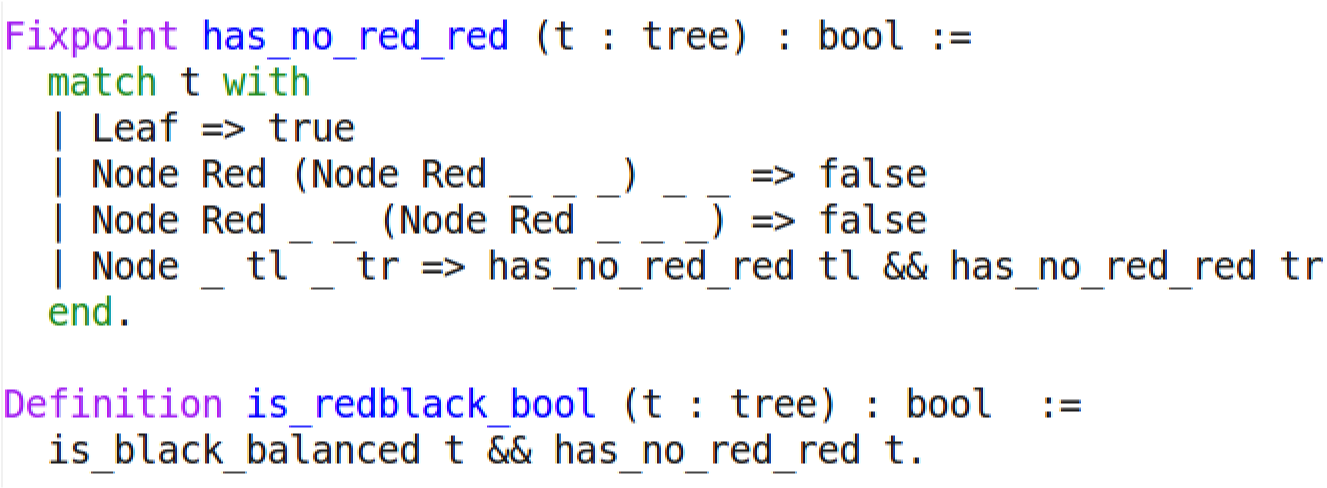
```
(* Red-Black Tree invariant: declarative definition *)
Inductive is_redblack' : tree -> color -> nat -> Prop :=
  | IsRB_leaf: forall c, is_redblack' Leaf c 0
  | IsRB_r: forall n tl tr h,
             is_redblack' tl Red h -> is_redblack' tr Red h ->
             is_redblack' (Node Red tl n tr) 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 *)
```

Property Checker

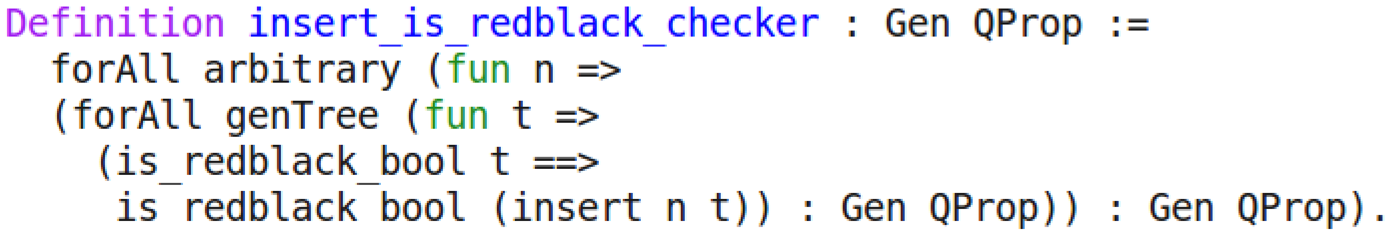
(efficiently executable definitions)



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



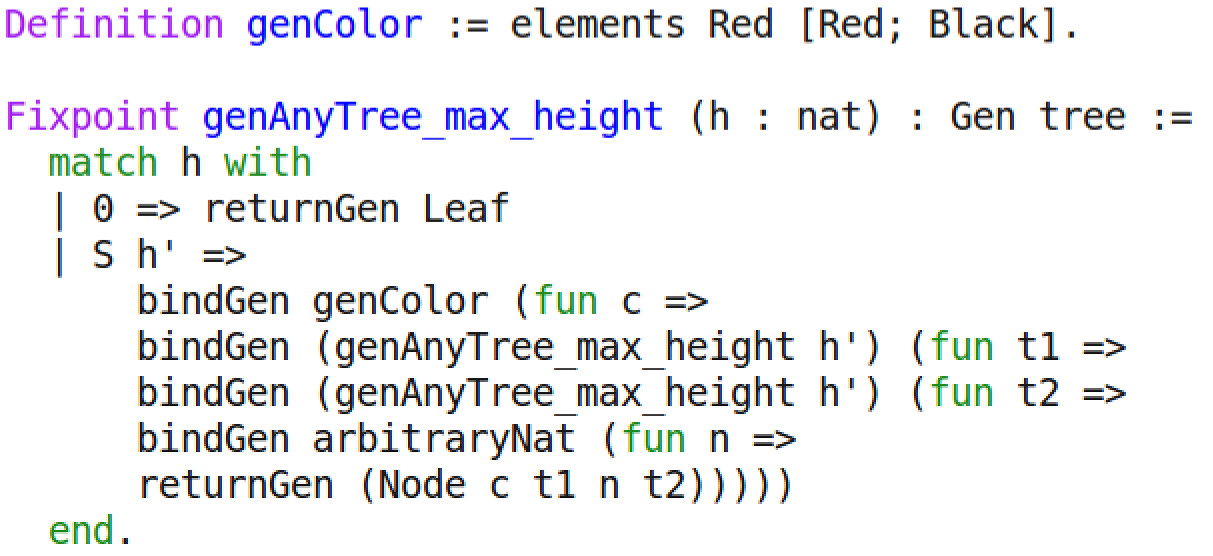
```
Fixpoint has_no_red_red (t : tree) : bool :=
  match t with
  | Leaf => true
  | Node Red (Node Red _ _ _) _ _ => false
  | Node Red _ _ (Node Red _ _ _) => false
  | Node _ tl _ tr => has_no_red_red tl && has_no_red_red tr
  end.

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).
```

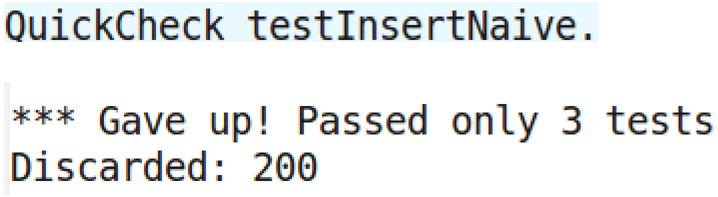
Generator for Arbitrary Trees

(this could one day be produced automatically)



```
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.
```

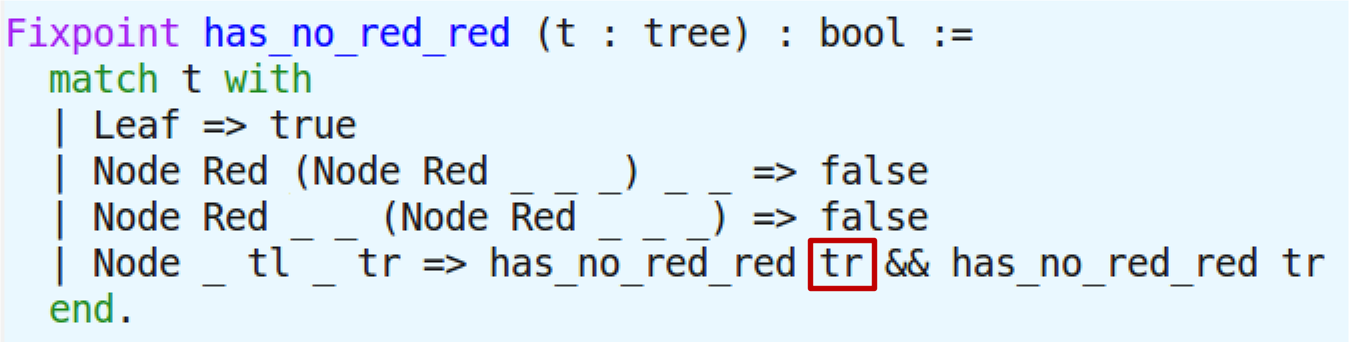
```
Definition genAnyTree : Gen tree := sized genAnyTree_max_height.
```



```
QuickCheck testInsertNaive.
```

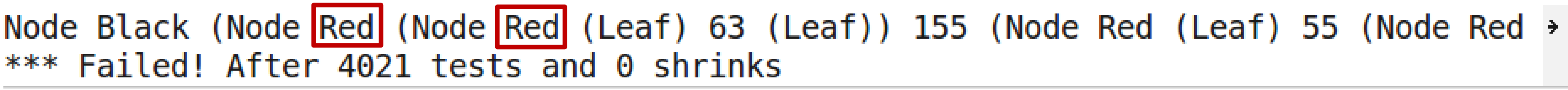
```
*** Gave up! Passed only 3 tests
Discarded: 200
```

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.
```

```
QuickCheck testInsertNaive.
```



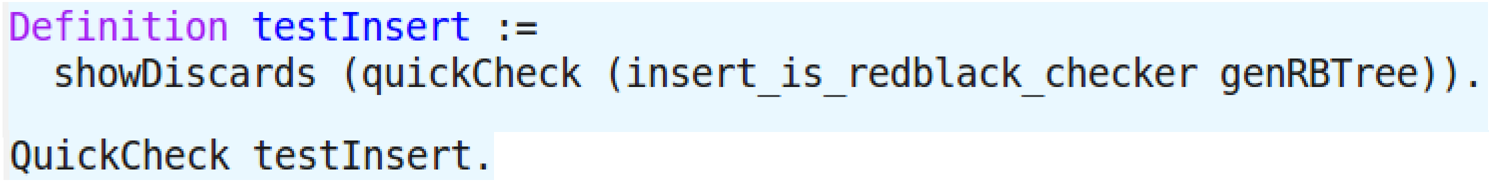
```
Node Black (Node Red (Node Red (Leaf) 63 (Leaf)) 155 (Node Red (Leaf) 55 (Node Red
*** Failed! After 4021 tests and 0 shrinks
```

Generator for Red-Black Trees

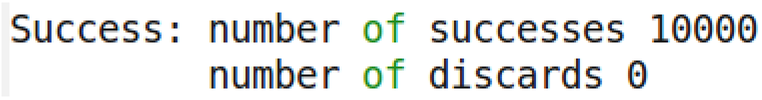
(handwritten property-based generator)

```
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



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

QuickCheck testInsert.
```



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

in less than 4 seconds

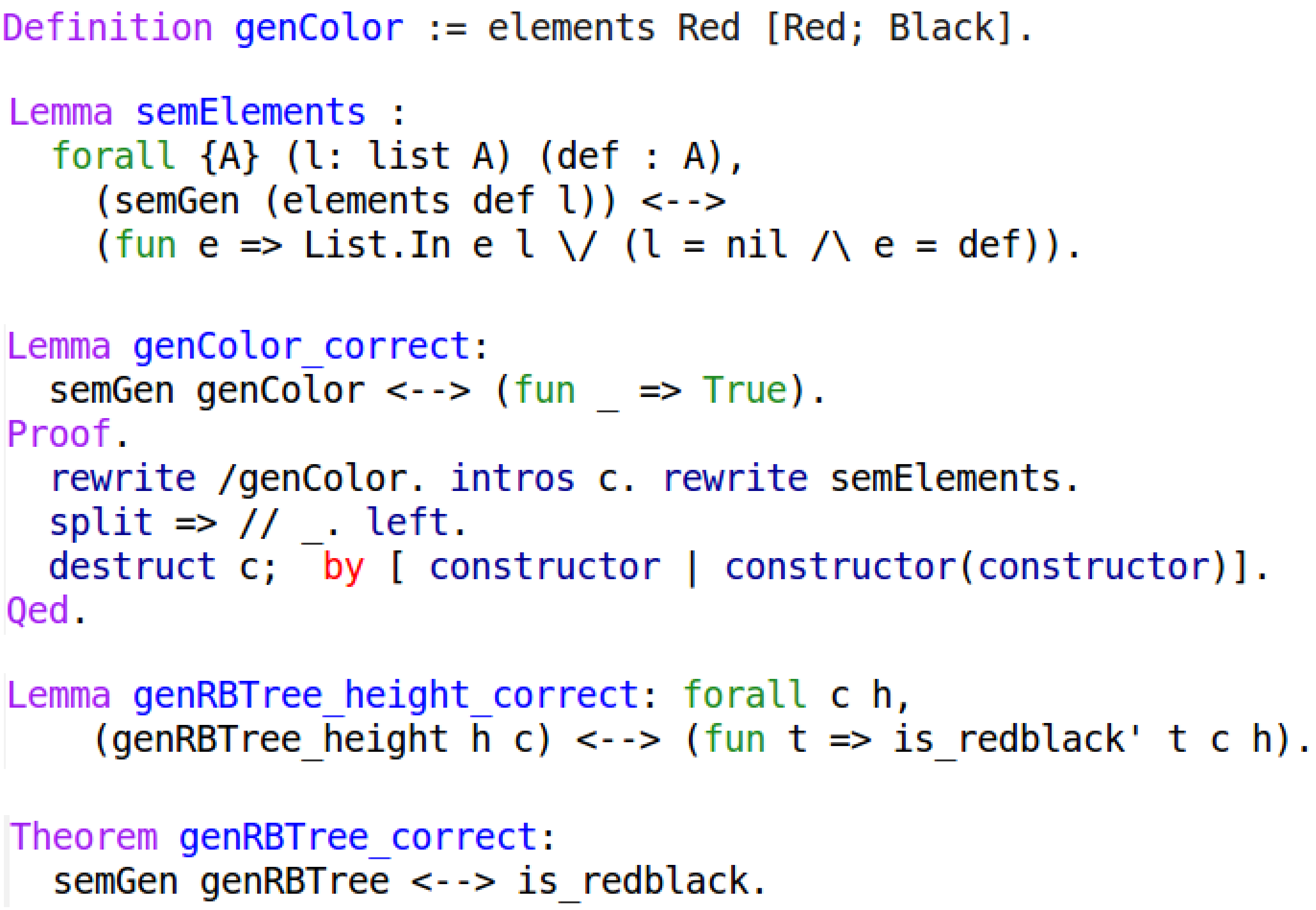
Zoe Paraskevopoulou
(ENS Cachan,
Inria intern last summer)

Cătălin Hrițcu
(Inria)

Are we testing the right property?

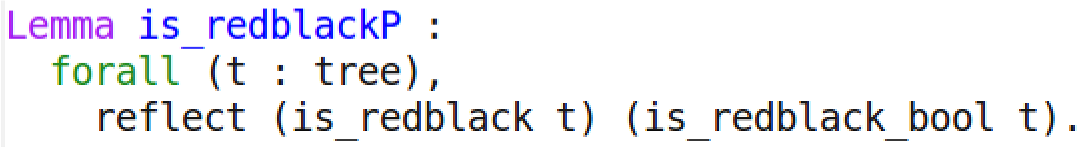
VERIFIED PROPERTY-BASED TESTING

Proving correctness of generators

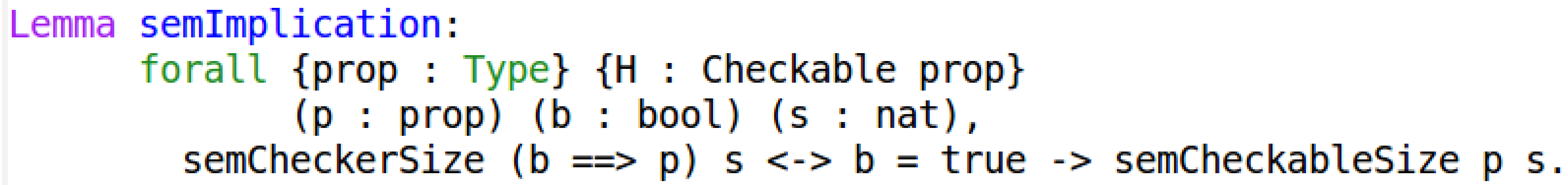


```
Definition genColor := elements Red [Red; Black].

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


Lemma genColor_correct:
  semGen genColor <--> (fun _ => True).
Proof.
  rewrite /genColor. intros c. rewrite semElements.
  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).

Theorem genRBTree_correct:
  semGen genRBTree <--> is_redblack.
```

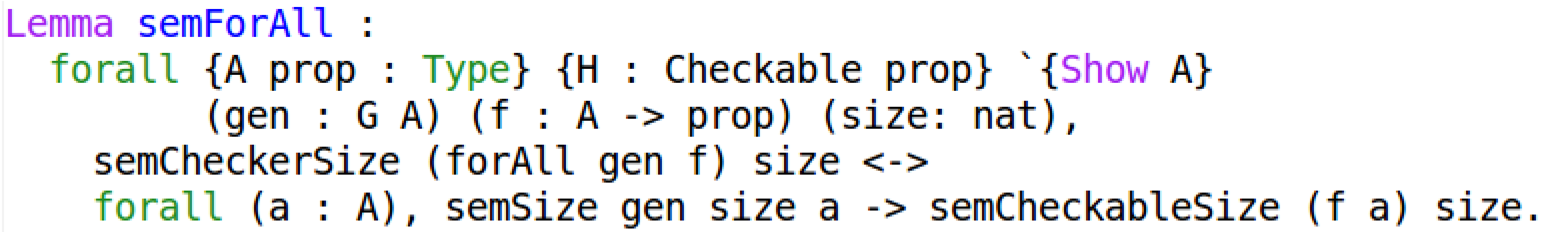
Proving correctness of checkers



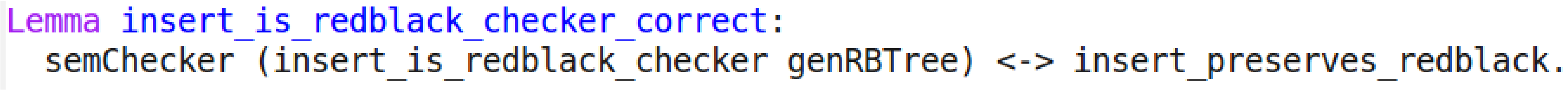
```
Lemma is_redblackP :
  forall (t : tree),
    reflect (is_redblack t) (is_redblack_bool t).
```



```
Lemma semImplication:
      forall {prop : Type} {H : Checkable prop}
             (p : prop) (b : bool) (s : nat),
        semCheckerSize (b ==> p) s <-> b = true -> semCheckableSize p s.
```



```
Lemma semForAll :
  forall {A prop : Type} {H : Checkable prop} `{Show A}
         (gen : G A) (f : A -> prop) (size: nat),
    semCheckerSize (forAll gen f) size <->
    forall (a : A), semSize gen size a -> semCheckableSize (f a) size.
```



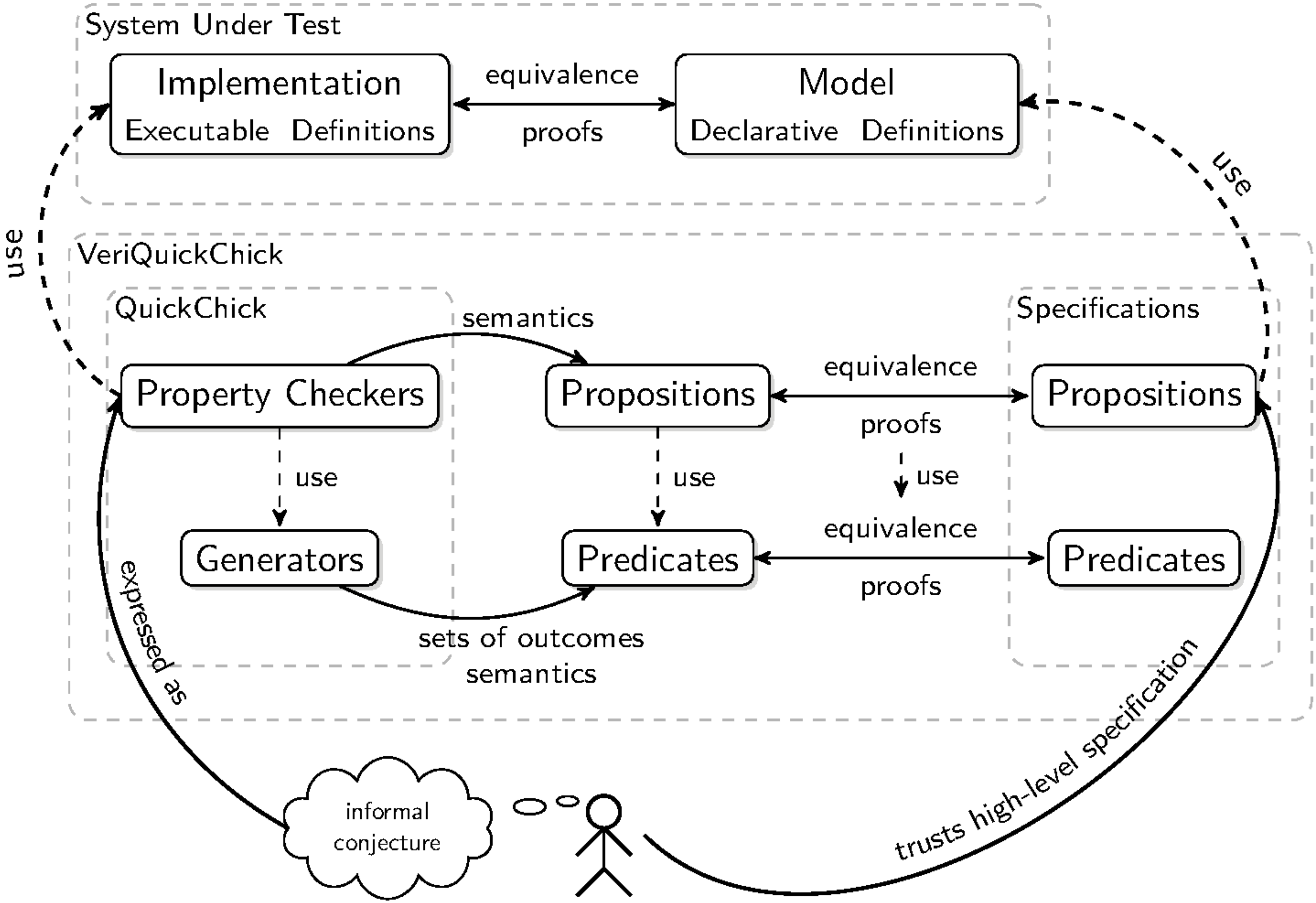
```
Lemma insert_is_redblack_checker_correct:
  semChecker (insert_is_redblack_checker genRBTree) <-> insert_preserves_redblack.
```

Set of outcomes semantics

- semantics of a generator is a set
    - intuitively containing the values that can be generated with >0 probability
- semantics of a checker is a Coq proposition

Formally we define

```
Definition Ensemble (A : Type) := A -> Prop.

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

Definition semSize {A : Type} (g : Gen A) (size : nat) : Ensemble A :=
  fun a => exists seed, (unGen g) seed size = a.

Definition semGen {A : Type} (g : Gen A) : Ensemble A :=
  fun a => exists size, semSize g size a.


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

Definition Checker : Type := Gen QProp.

Definition semChecker (P : Checker) : Prop :=
  forall s qp, semSize P s qp -> success qp = true.
```



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

equivalence

Generators

Predicates

Predicates

proofs

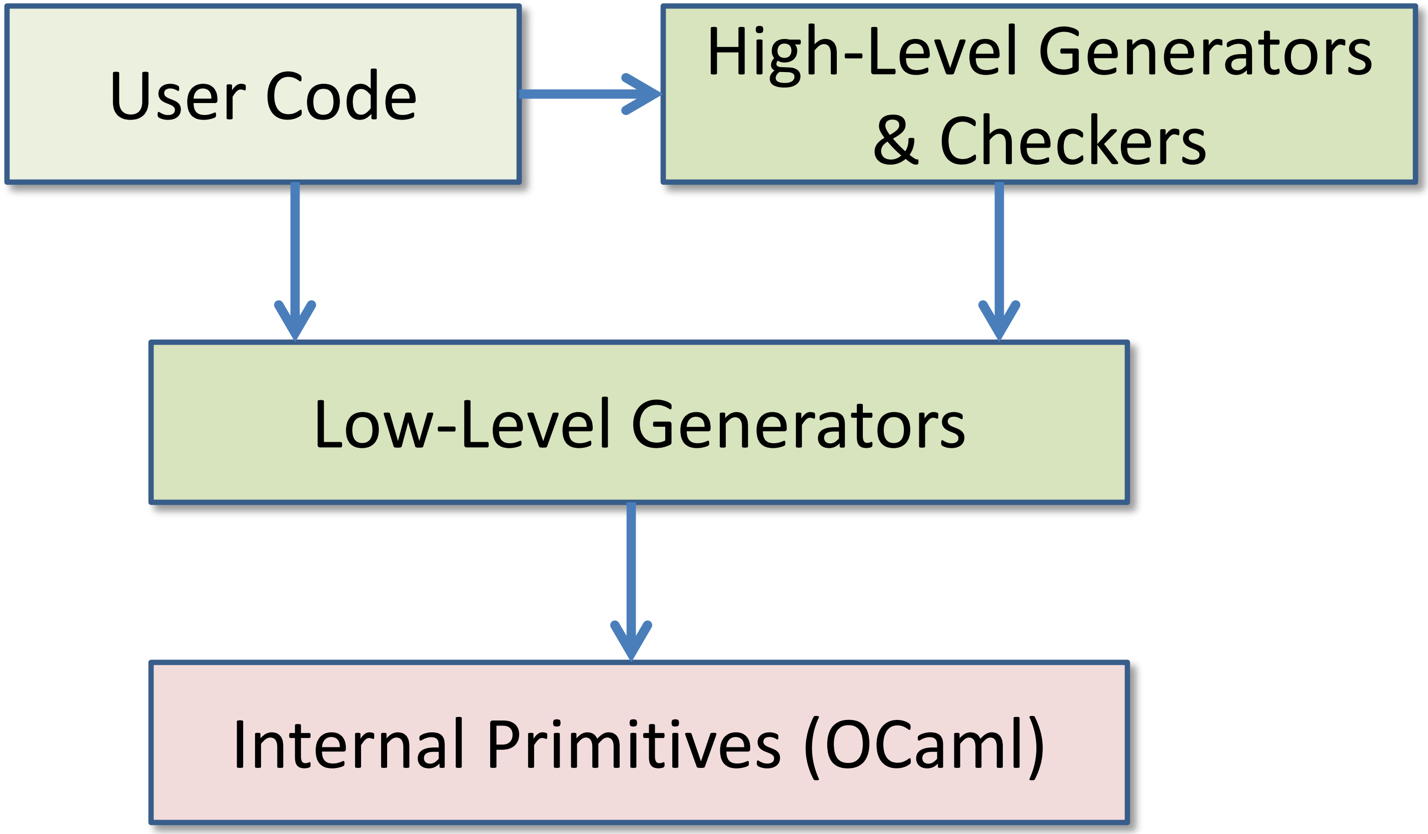
expressed as

sets of outcomes semantics

informal conjecture

trusts high-level specification

QuickChick/Proof Organization



User Code

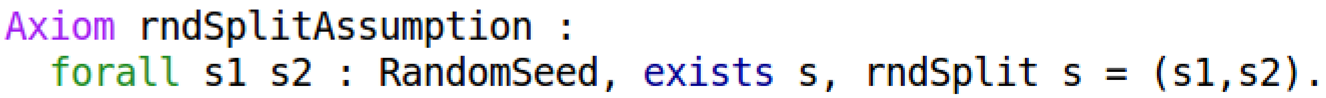
High-Level Generators & Checkers

Low-Level Generators

Internal Primitives (OCaml)

Internal Primitives & Axiom(s)

- random seed type + 8 primitive functions written only in OCaml and only *assumed* in Coq
- 5 axioms about these primitive functions
  - 4 of them would disappear if we implemented a splittable random number generator in Coq
  - ***remaining axiom is inherent to our abstraction!***



```
Axiom rndSplitAssumption :
  forall s1 s2 : RandomSeed, exists s, rndSplit s = (s1,s2).
```

    - makes the type RandomSeed infinite in Coq,
      while in OCaml it is finite (seeds are bounded integers)
  - we assume *real randomness* (an oracle) in the proofs, but can only implement *pseudo-randomeness*

Lemmas for Low-Level QC Generators (10)

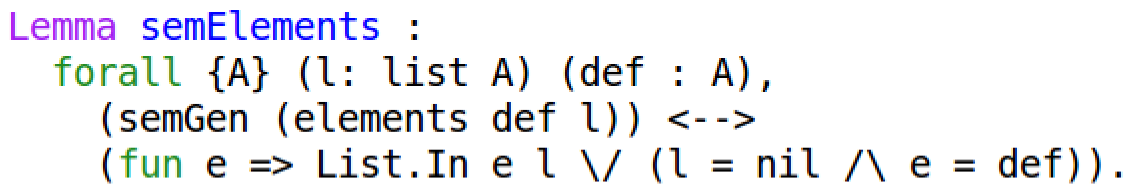
- they rely on primitives and concrete representation of Gen

```
Lemma semReturnSize : forall A (x : A) (size : nat),
    semSize (returnGen x) size <--> eq x.

Lemma semBindSize : forall A B (g : G A) (f : A -> G B) (size : nat),
  semSize (bindGen g f) size <--> (fun b => exists a, (semSize g size) a /\
                                                      (semSize (f a) size) b).
```

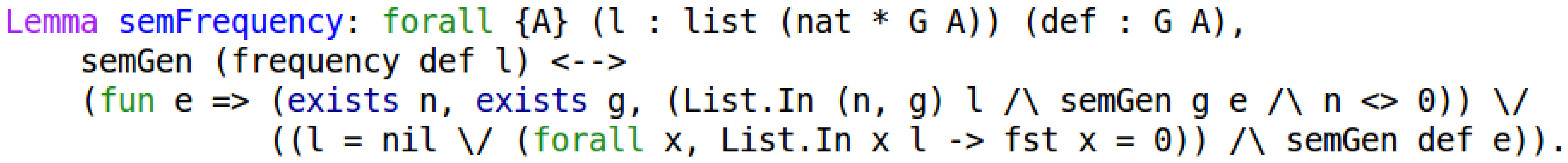
- bind proof crucially relies on axiom about rndSplit
- we can’t abstract over the sizes (existentially quantify)

```
Lemma semSizedSize :
  forall A (f : nat -> G A),
    semGen (sized f) <--> (fun a => exists n, semSize (f n) n a).

Lemma semResize :
  forall A (n : nat) (g : G A), semGen (resize n g) <--> semSize g n.
```

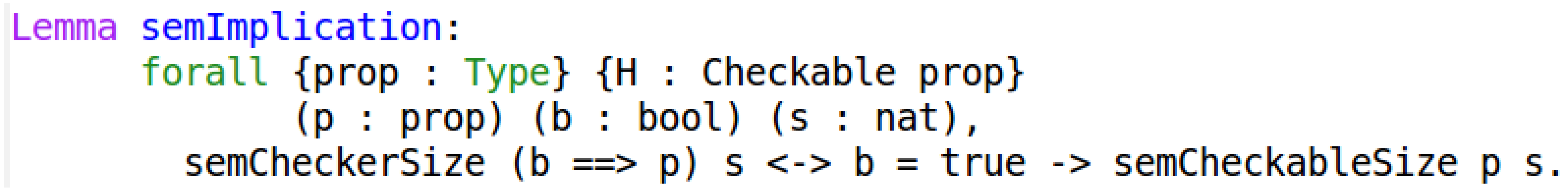
High-Level Generators & Checkers (12)



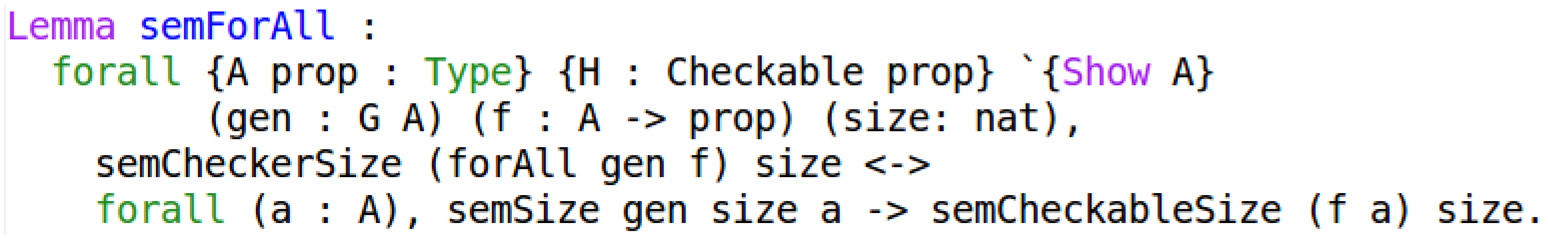
```
Lemma semElements :
  forall {A} (l: list A) (def : A),
    (semGen (elements def l)) <-->
    (fun e => List.In e l \/ (l = nil /\ e = def)).
```



```
Lemma semFrequency: forall {A} (l : list (nat * G A)) (def : G A),
    semGen (frequency def l) <-->
    (fun e => (exists n, exists g, (List.In (n, g) l /\ semGen g e /\ n <> 0)) \/
                ((l = nil \/ (forall x, List.In x l -> fst x = 0)) /\ semGen def e)).
```



```
Lemma semImplication:
      forall {prop : Type} {H : Checkable prop}
             (p : prop) (b : bool) (s : nat),
        semCheckerSize (b ==> p) s <-> b = true -> semCheckableSize p s.
```



```
Lemma semForAll :
  forall {A prop : Type} {H : Checkable prop} `{Show A}
         (gen : G A) (f : A -> prop) (size: nat),
    semCheckerSize (forAll gen f) size <->
    forall (a : A), semSize gen size a -> semCheckableSize (f a) size.
```

Summary

- Coq framework for verified PBT
- Integrated in QuickChick
    - https://github.com/QuickChick
- Reasoning about sets of outcomes
- The first verified QuickCheck implementation
- Examples: red-black trees and noninterference
- Modular, scalable, minimal code changes

Future Work

- More proof automation and infrastructure
  - – changing to efficient data representations
  - – SMT-based verification for sets of outcomes
- Verify property-based generator language
- Probabilistic verification
- Splittable RNG in Coq
- Try to reduce testing cost, now significant
  - – break even point very much problem-specific

THANK YOU

Code at https://github.com/QuickChick