Foundational Property-Based Testing

Cătălin Hriţcu

INRIA Paris

Inria
INVENTORS FOR THE DIGITAL WORLD

My research projects

- **Micro-Policies**: Formally Verified, Hardware-Assisted, Tag-Based Security Monitors

- **F***: ML-like Programming Language + Verification System + Proof Assistant

- **Property-Based Testing for Verified Languages & Systems**
  - QuickChick & Foundational & Luck & Mutants & Testing IFC ...

this talk

Collaborators

Arthur Azevedo de Amorim
(UPenn, Inria intern 2014)

Maxime Dénès
(Inria)

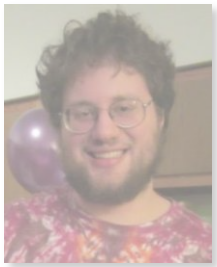
John Hughes
(Chalmers)

Cătălin Hriţcu
(Inria)

Leo Lampropoulos
(UPenn)

Zoe Paraskevopoulou
(Princeton,
Inria intern 2014)

Benjamin Pierce
(UPenn)



Antal Spector-Zabusky
(UPenn)

Dimitris Vytiniotis
(MSR Cambridge)

Li-yao Xia
(ENS Paris,
Inria intern
2015)

Verification is **expensive**

- When designing and verifying languages and systems, ***most enlightenment comes from counterexamples***
- *but* ***finding counterexamples via failed proofs is very costly***
- ***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 ...***
- ***our hope:*** property-based testing can help

A QUICK INTRODUCTION TO PBT USING 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 potentially 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 seconds

QuickChick

- Coq clone of Haskell QuickCheck
- Mostly written in Coq, extracted to OCaml
- Code at https://github.com/QuickChick

Larger case study on

TESTING NONINTERFERENCE, QUICKLY

[ICFP 2013 and beyond]

Can we quickcheck **noninterference**?

[ICFP 2013]

- **Context**
  - designing **real machine with dynamic IFC** (>100 instructions)
- **Experiment**
  - **very simple stack machine** (10 instructions)
  - standard **end-to-end noninterference property**
  - manually **introduced 14 plausible IFC errors**, and **measured how fast they are found**
- **Encouraging results**
  - however, **not using QuickCheck naïvely**

3 secret ingredients

1. **Fancy program generation strategies**
   - $s_1 \approx s_2$ – generate s_1 then vary secrets to get $s_2 \approx s_1$
   - distributions, instruction sequences, smart integers
   - best one: "generation by execution"
     - **19 instructions counterexample takes minutes to find**
2. **Strengthening the tested property**
   - best one: "unwinding conditions" (inductive invariant)
     - **all errors found in milliseconds, even with simple generation**
   - requires finding stronger invariants, like for proving
3. **Fancy counterexample shrinking**

Rather simple custom generator

```
frequency $
  [ (1,  pure Noop) ] ++
  [ (1,  pure Halt) ] ++
  [ (10, pure Add) | nstk >= 2 ] ++
  [ (10, Push <$> lint) ] ++
  [ (10, pure Pop) | nstk >= 1 ] ++
  [ (20, pure Store) | nstk >= 2
                     , absAdjustAddr vtop `isIndex` mem ] ++
  [ (20, pure Load) | nstk >= 1
                    , absAdjustAddr vtop `isIndex` mem ] ++
  [ (10, liftM2 Call (choose (0, (nstk-1) `min` maxArgs)) arbitrary)
                     | nstk >= 1
                     , cally ] ++
  [ (20, liftM Return arbitrary) | Just r <- [ fmap astkReturns $
                                               find (not . isAData) stk]
                                 , nstk >= if r then 1 else 0
                                 , cally ] ++
  [ (10, pure Jump) | nstk >= 1
                    , jumpy ] ++
```

Guarded (add needs 2 arguments)

Store only to valid address

Return only after call

Biased generator

....

Scaling this up

[JFP 2015 submission]

- More complex register machine
  - with dynamic memory allocation
- Fancier IFC features:
  - public first-class labels, flow sensitive analysis
- Used property-based testing to
  - design novel highly-permissive dynamic IFC mechanism
  - discover complex invariants of noninterference proof
  - Coq proof took only ~2 weeks afterwards and found only one error
  - prior (paper) proof attempt timed out after ~3 weeks of work
- Extra ingredient: **4. polarized mutation testing**
  - 33 mutants: covering all missing taints / check bugs

PBT is not very reliable

- PBT is not push button ... users has to write testing 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
- **how can we find the bugs in our testing?**
- when testing finds no bugs ... is the property true?
  - **are we are testing things well?**
  - **are we even testing the right property?**

How can make PBT more reliable?

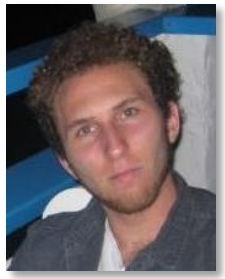
- **Gather statistics**
  - discard rates, distributions, coverage, etc.
  - can discover bugs if one looks at the “right” thing
- **Add bugs** and make sure they are found
  - **polarized mutation testing**

- **Formally verify** the testing code
  - **foundational property-based testing**

- **Better languages** for writing testing code
  - **Luck: new language for generators**

Cătălin Hriţcu
(Inria)

Maxime Dénès
(Inria)



Leo Lampropoulos
(UPenn)

Benjamin Pierce
(UPenn)

Are we testing things right? Introducing true bugs!

POLARIZED MUTATION TESTING

Mutation testing

- Automatically introduce bugs
  - **test the testing infrastructure** (e.g. the generator)
  - in ICFP 2013 experiments we added bugs manually
    - does not scale, tedious and turns code into spaghetti
  - enumerate all missing taints and missing checks
    - especially easy when IFC split into separate “rule table”

Rule table

	Allow	**Result**	**PC**
OpLab	TRUE	BOT	LabPC
OpMLab	TRUE	Lab1	LabPC
OpPcLab	TRUE	BOT	LabPC
OpBCall	TRUE	JOIN Lab2 LabPC	JOIN Lab1 LabPC
OpBRet	LE (JOIN Lab1 LabPC) (JOIN Lab2 Lab3)	Lab2	Lab3
OpFlowsTo	TRUE	JOIN Lab1 Lab2	LabPC
OpLJoin	TRUE	JOIN Lab1 Lab2	LabPC
OpPutBot	TRUE	BOT	LabPC
OpNop	TRUE	__	LabPC
OpPut	TRUE	BOT	LabPC
OpBinOp	TRUE	JOIN Lab1 Lab2	LabPC
OpJump	TRUE	__	JOIN LabPC Lab1
OpBNZ	TRUE	__	JOIN Lab1 LabPC
OpLoad	TRUE	Lab3	JOIN LabPC (JOIN Lab1 Lab2)
OpStore	LE (JOIN Lab1 LabPC) Lab2	Lab3	LabPC
...	...	...	...

Results encouraging

- Our **generator had tons of bugs**
  - could only kill 9 out of 51 mutants (17.6%)!
- **Finding and fixing generator bugs**
  - gathering statistics, constructing counterexamples by hand
  - fixing one generator bug usually killed many more mutants
  - sometimes found extra bugs in un-mutated artifact & property
- After a couple of days **only live 2 mutants**
  - for which we still couldn't find counterexamples by hand
  - we applied these mutantions ... started proving
- **Mutation testing gamifies invariant finding**
  - to the point it's actually fun and addictive!

Mutant game (final output)

```
./Extracted
Fighting 52 mutants
Killed mutant 0 (1 frags)
Killed mutant 1 (2 frags)
Killed mutant 2 (3 frags)
Killed mutant 3 (4 frags)
Killed mutant 4 (5 frags)
Killed mutant 5 (6 frags)
Killed mutant 6 (7 frags)
Killed mutant 7 (8 frags)
Killed mutant 8 (9 frags)
Killed mutant 9 (10 frags)
Killed mutant 10 (11 frags)
Killed mutant 11 (12 frags)
Killed mutant 12 (13 frags)
Killed mutant 13 (14 frags)
Killed mutant 14 (15 frags)
Killed mutant 15 (16 frags)
Killed mutant 16 (17 frags)
Killed mutant 17 (18 frags)
Killed mutant 18 (19 frags)
Killed mutant 19 (20 frags)
Killed mutant 20 (21 frags)
Killed mutant 21 (22 frags)
Killed mutant 22 (23 frags)
Killed mutant 23 (24 frags)
Killed mutant 24 (25 frags)
Killed mutant 25 (26 frags)
Killed mutant 26 (27 frags)
Killed mutant 27 (28 frags)
Killed mutant 28 (29 frags)
Killed mutant 29 (30 frags)
Killed mutant 30 (31 frags)
Killed mutant 31 (32 frags)
Killed mutant 32 (33 frags)
Killed mutant 33 (34 frags)
Killed mutant 34 (35 frags)
Killed mutant 35 (36 frags)
Killed mutant 36 (37 frags)
Killed mutant 37 (38 frags)
Missed mutant [38] (38 frags)
Missed mutant [39] (38 frags)
Killed mutant 40 (39 frags)
Killed mutant 41 (40 frags)
Killed mutant 42 (41 frags)
Killed mutant 43 (42 frags)
Killed mutant 44 (43 frags)
Killed mutant 45 (44 frags)
Killed mutant 46 (45 frags)
Killed mutant 47 (46 frags)
Killed mutant 48 (47 frags)
Killed mutant 49 (48 frags)
Killed mutant 50 (49 frags)
Killed mutant 51 (50 frags)
```

So why did this work so well?

- Yes, human is in the loop (debugging, finding counterexamples)
  - but **we don't waste human cycles**
- **Each unkilled mutant thought us something**
  - either exposed real bugs in the testing
  - or was better than the original (more permissive)
- **Property-based: strengthening a supposedly "tight" property**
- **This is usually not the case for mutation testing**
  - purely syntactic mutations (replace "+" by "-")
  - human cycles wasted on silly ("equivalent") mutants that don't break the tested property
  - kill count just alternative to code coverage metrics, never 100%

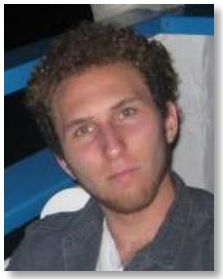
Polarized mutation testing

- Generalizing this technique beyond IFC
  - we eventually want a framework
- Started with STLC experiment
  - break progress by strengthening the step relation (e.g. dropping whole stepping rules)
  - break preservation
    - by strengthening positive occurrence of typing relation
    - or by weakening negative occurrence of typing relation
    - or by weakening (negative occurrence of) step relation
  - no-shadowing bug in fancy generator for well-typed terms
- We broke tail call optimization in CompCert (made all calls tail), found that CSmith couldn't find the problem
- This works well in practice, but hard to formalize

Zoe Paraskevopoulou
(Princeton,
Inria intern 2014)

Cătălin Hriţcu
(Inria)

Maxime Dénès
(Inria)



Leo Lampropoulos
(UPenn)

Benjamin Pierce
(UPenn)

Are we testing the right property?

FOUNDATIONAL PROPERTY-BASED TESTING

[ITP 2015]

- 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. Testing code easy to get wrong
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 Foundational 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

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.
```

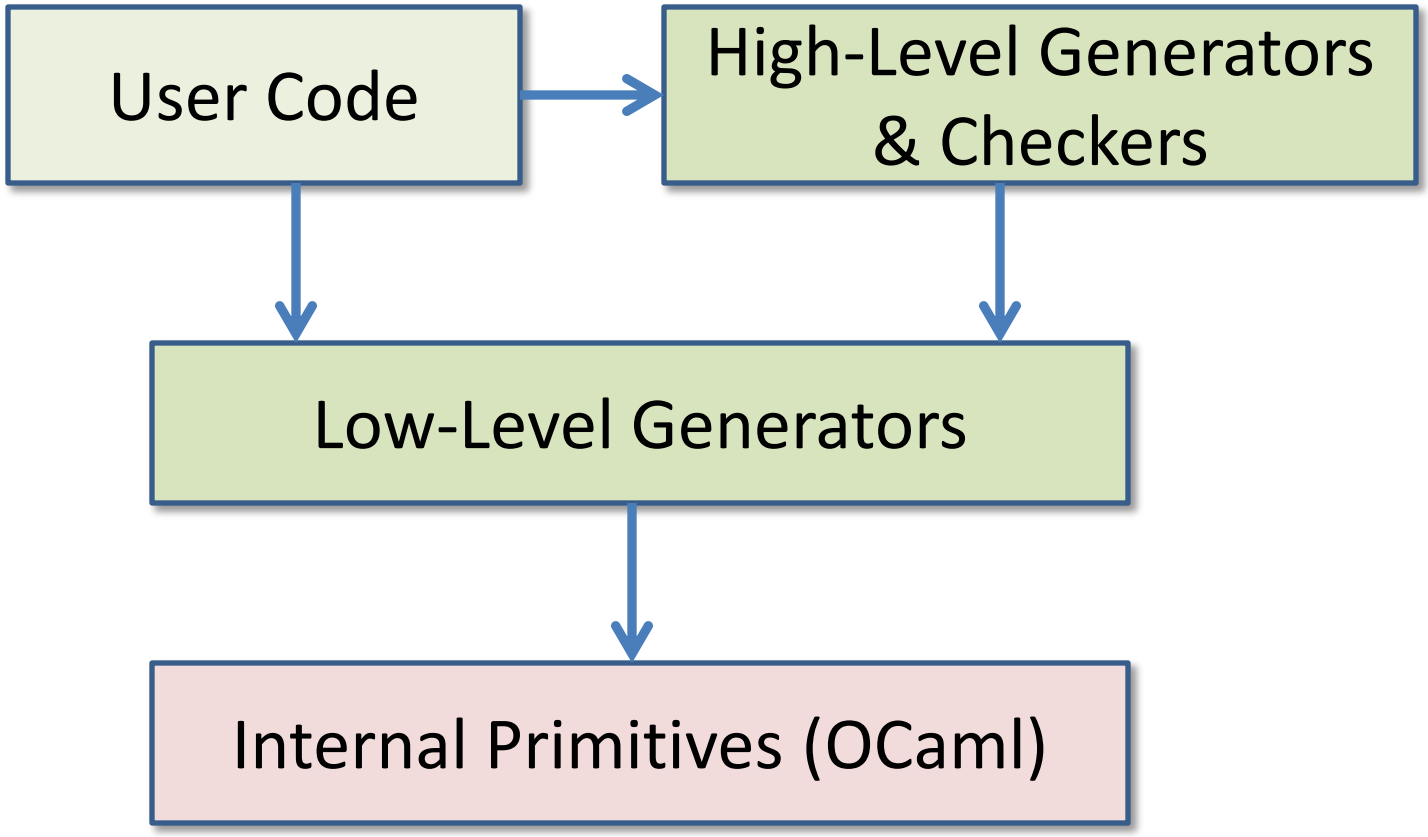
Set of outcomes semantics

- 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.
```

QuickChick/Proof Organization



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*

Size Abstraction

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

```
1  Lemma semBindSize A B (g : G A) (f : A → G B) (s : nat) :
2    semGenSize (bindGen g f) s ≡
3    \bigcup_(a in semGenSize g s) semGenSize (f a) s.
```

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

```
1  Lemma semBindSizeMonotonic :
2    ∀ {A B} (g : G A) (f : A → G B)
3          `{ SizeMonotonic _ g} `{∀ a, SizeMonotonic (f a)},
4      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

Summary

- Coq framework for foundational 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

Cătălin Hriţcu
(Inria)

John Hughes
(Chalmers)

Leo Lampropoulos
(UPenn)

Zoe Paraskevopoulou
(Princeton,
Inria intern 2014)

Benjamin Pierce
(UPenn)

Li-yao Xia
(ENS Paris,
Inria intern
2015)

Better languages for writing testing code

LUCK: LANGUAGE FOR GENERATORS

Luck

- Functional programming language
- Expressions have **dual semantics**
  - checkers when all variables are fully instantiated
  - generators when variables are partially defined
- Puts together:
  - lazy instantiation
    - “narrowing” from functional logic programming
  - constraint propagation for integers
  - local control over probability distribution

Red-black trees in Luck

```
sig isRBT' :: Int -> Int -> Int -> Color -> RBT Int -> Bool
fun isRBT' {h @i} {low @i} {high @i} c t =
  if h == 0 then
      case c of
        | Red -> isLeaf t
        | Black ->
           case t of
             | Leaf -> True
             | Node k x l r -> [| x | low < x && x < high |]
                               && isLeaf l && isLeaf r && isBlack k
           end
      end
  else ...
```

Luck is work in progress ...

- Experimental results vary: 1.5x-50x overhead
- Semantics is challenging
  - generalize from bool to arbitrary result types
  - preserve sharing between values
- Final meta theorems we're aiming for
  - soundness & completeness
    - inspired from foundational PBT work
  - but also capturing probabilities and backtracking

Future Work

- A lot of this is ongoing / unfinished work:
  - Luck, polarized mutation
- Foundational PBT
  - SMT-based verification for sets of outcomes
- Further reduce testing cost
  - more automation (e.g. for inductives)
  - hopefully as certificate producing metaprograms
- Integration:
  - bring Luck to Haskell and Coq

THANK YOU




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.
```