QuickChick: Property-Based Testing in Coq

POPL 2019
Tutorial Fest
14/01/2019

Leonidas Lampropoulos
Benjamin C. Pierce
QuickChick: Property-Based Testing in Coq

Leonidas Lampropoulos
Benjamin C. Pierce
High-level view of workflow

Theorem preservation :=

\[ \text{forall } e \, T \, \Gamma, \]
\[ \Gamma \vdash e : T \rightarrow e \Rightarrow e' \rightarrow \Gamma \vdash e' : T. \]
High-level view of workflow

\[
\text{Theorem preservation := } \\
\forall e \ T \ \Gamma, \\
\Gamma \vdash e : T -> e =\Rightarrow e' -> \Gamma \vdash e' : T.
\]
Theorem preservation :=
for all e T \Gamma,
\Gamma |- e : T -> e \Rightarrow e' -> \Gamma |- e' : T.
A better workflow

Theorem preservation :=
  \forall e \in T. \Gamma,
  \Gamma \vdash e : T \implies e \Rightarrow e' \implies \Gamma \vdash e' : T.
A better workflow

Theorem preservation :=

\[ \forall e \in T, \Gamma, \quad \Gamma |- e : T \rightarrow e \Rightarrow e' \rightarrow \Gamma |- e' : T. \]
A better workflow

Theorem preservation :=
for all e : T, Γ,
Γ |- e : T ⇒ e ⇒ e' ⇒ Γ |- e' : T.
A better workflow

Theorem preservation :=
for all e T Γ,
Γ |- e : T -> e => e' -> Γ |- e' : T.
Bugs are everywhere

Testing  Verification
Testing

• Excellent at discovering bugs
  • [CSmith] : More than 400 bugs in C compilers (GCC, LLVM)
  • [Palka et al. ‘11] : Bugs in GHC’s strictness analyzer

• Cannot guarantee their absence
  “Testing shows the presence, not the absence of bugs” - Dijkstra
Verification

• Strong formal guarantees
• Recent success stories
  • [CompCert] : Verified optimizing C compiler
  • [CertiKOS] : Extensible architecture for certified OS kernels
• ...but still very expensive
• Already present in many proof assistants
  • Isabelle [Berghofer 2004, Bulwahn 2012]
  • Agda [Dybjer et al 2003]
  • ACL2 [Chamarthi et al 2011]
Testing in Coq

• Already present in many proof assistants
  • Isabelle [Berghofer 2004, Bulwahn 2012]
  • Agda [Dybjer et al 2003]
  • ACL2 [Chamarthi et al 2011]

• QuickChick
  • Coq port of Haskell QuickCheck
  • On steroids! [ITP 2015, POPL 2018]
Roadmap

- Property-based testing overview
- Case Study: Expression Compiler
  - Typeclasses and automation
  - Properties
  - Batch execution
  - Mutation Testing
- Beyond Automation - Generators
- Open Research!
Overview of property-based testing

Theorem preservation :=

\[
\forall e \Gamma, \quad 
\Gamma \vdash e : T \Rightarrow e \Rightarrow e' \Rightarrow \Gamma \vdash e' : T.
\]

QuickChick preservation.
Overview of property-based testing

Theorem preservation :=

\[\forall e \in T \Gamma, \Gamma \vdash e : T \Rightarrow e \Rightarrow e' \Rightarrow \Gamma \vdash e' : T.\]

QuickChick preservation.
Overview of property-based testing

Theorem preservation :=

\[
\forall e \Gamma, \\
\Gamma \vdash e : T \rightarrow e \Rightarrow e' \rightarrow \Gamma \vdash e' : T.
\]

QuickChick preservation.
Overview of property-based testing

Theorem preservation :=
forall e T Γ,
Γ |- e : T -> e => e' -> Γ |- e' : T.

QuickChick preservation.
Theorem preservation :=
  \( \forall e \ T \ \Gamma, \Gamma |- e : T \Rightarrow e \Rightarrow e' \Rightarrow \Gamma |- e' : T. \)

QuickChick preservation.
Overview of property-based testing

Theorem preservation :=
\[
\forall e \ T \ \Gamma, \\
\Gamma \vdash e : T \rightarrow e \Rightarrow e' \rightarrow \Gamma \vdash e' : T.
\]

QuickChick preservation.
Demo time!
Arithmetic Expressions

42

17 + 25

(22 - 1) * 2
Arithmetic Expressions

42

17 + 25

(22 - 1) * 2

Inductive exp : Type :=
| ANum : nat → exp
| APlus : exp → exp → exp
| AMinus : exp → exp → exp
| AMult : exp → exp → exp.
Arithmetic Expressions

\[ 42 \]
\[ 17 + 25 \]
\[ (22 - 1) \times 2 \]

\[ \text{ANum} \, 42 \]
\[ \text{APlus} \, (\text{ANum} \, 17) \, (\text{ANum} \, 25) \]
\[ \text{AMult} \, (\text{AMinus} \, (\text{ANum} \, 22) \, (\text{ANum} \, 1)) \, (\text{ANum} \, 2) \]

\[
\text{Inductive exp : Type :=}
\mid \text{ANum : nat → exp}
\mid \text{APlus : exp → exp → exp}
\mid \text{AMinus : exp → exp → exp}
\mid \text{AMult : exp → exp → exp.}
\]
Inductive exp : Type :=
  | ANum : nat → exp
  | APlus : exp → exp → exp
  | AMinus : exp → exp → exp
  | AMult : exp → exp → exp.
Fixpoint show_exp e : string :=
  match e with
  | ANum e => “ANum “ ++ show_nat e
  | APlus e1 e2 =>
    “APlus (“ ++ show_exp e1 ++ “) (“ ++ show_exp e2 ++ “)”
  | ...

Inductive exp : Type :=
  | ANum : nat → exp
  | APlus : exp → exp → exp
  | AMinus : exp → exp → exp
  | AMult : exp → exp → exp.
Fixpoint show_exp e : string :=
match e with
| ANum e => "ANum " ++ show_nat e
| APlus e1 e2 =>
  "APlus (" ++ show_exp e1 ++ ") (" ++ show_exp e2 ++ ")"
| ...

Explicitly remember how to print nats

Inductive exp : Type :=
| ANum : nat → exp
| APlus : exp → exp → exp
| AMinus : exp → exp → exp
| AMult : exp → exp → exp.
Class Show (A : Type) := \{ show : A -> string \}.  

The Show TypeClass
The Show TypeClass

Class Show (A : Type) := { show : A -> string }. 
Class Show (A : Type) := { show : A -> string }.

Instance show_exp : Show exp :=
{| show e :=
    match e with
    | ANum e => "ANum " ++ show_nat e
    | APlus e1 e2 =>
      "APlus (" ++ show e1 ++ ") (" ++ show e2 ++ ")"
    | ...
    |}. 
Class Show (A : Type) := { show : A -> string }.

Instance show_exp : Show exp :=
{| show e :=
  match e with
  | ANum e => “ANum “ ++ show e
  | APlus e1 e2 =>
    “APlus (“ ++ show e1 ++ “) (“ ++ show e2 ++ “)”
  | ...
|}.  

Class method
Class Show (A : Type) := { show : A -> string }.

Derive Show for exp.
Class Gen (A : Type) := { arbitrary : G A }.

Derive Arbitrary for exp.
Class Gen (A : Type) := { arbitrary : G A }.

Derive Arbitrary for exp.
Class Gen (A : Type) := { arbitrary : G A }.

Derive Arbitrary for exp.
Demo Time! - Sampling
Fixpoint eval (e : exp) : nat :=
match e with
  | ANum n => n
  | APlus e1 e2 => (eval e1) + (eval e2)
  | AMinus e1 e2 => (eval e1) - (eval e2)
  | AMult e1 e2 => (eval e1) * (eval e2)
end.
Optimizations:

- $0 + x \rightarrow x$
- $X + 0 \rightarrow x$
- $X * 0 \rightarrow 0$
- $X * 1 \rightarrow x$
- $X - 0 \rightarrow x$

...
An expression optimizer - Correctness

Optimizations:
0 + x → x
X + 0 → x
X * 0 → 0
X * 1 → x
X − 0 → x
...

e : exp
An expression optimizer - Correctness

Optimizations:

- $0 + x \rightarrow x$
- $x + 0 \rightarrow x$
- $x \cdot 0 \rightarrow 0$
- $x \cdot 1 \rightarrow x$
- $x - 0 \rightarrow x$
- ...

$e : \text{exp}$

eval

$n$
An expression optimizer - Correctness

Optimizations:

- $0 + x \rightarrow x$
- $X + 0 \rightarrow x$
- $X * 0 \rightarrow 0$
- $X * 1 \rightarrow x$
- $X - 0 \rightarrow x$
- ...

Diagram:

```
  e : exp
     ↓ optimize
   e' : exp
       ↓ eval
    n
```
An expression optimizer - Correctness

Optimizations:
0 + x → x
X + 0 → x
X * 0 → 0
X * 1 → x
X - 0 → x

...
Optimizations:
- $0 + x \rightarrow x$
- $X + 0 \rightarrow x$
- $X \cdot 0 \rightarrow 0$
- $X \cdot 1 \rightarrow x$
- $X - 0 \rightarrow x$
- ...

An expression optimizer - Correctness
The Stack Machine

Instructions:
- Add
- Push 2
- Mult

Stack:
- 1
- 20
The Stack Machine

Instructions:
- Add
- Push 2
- Mult

Stack:
- 1
- 20
The Stack Machine

Instructions:
Add Push 2 Mult

Stack:
1 20
The Stack Machine

Instructions:
- Push 2
- Mult

Stack: 21
The Stack Machine

Instructions:

Push 2  Mult

Stack

21
The Stack Machine

Instructions:

Stack

2

21

Mult
The Stack Machine

Instructions:

Stack

2

21

Mult
The Stack Machine

Instructions:

Stack

Mult

2

21
The Stack Machine

Instructions:
(20 + 1) * 2
(20 + 1) * 2

AMult (APlus (ANum 20) (ANum 1)) (ANum 2)
(20 + 1) * 2

\[ AMult \ (APlus \ (ANum\ 20) \ (ANum\ 1)) \ (ANum\ 2) \]
(20 + 1) * 2

\[ (20) \times 1 = 20 \times 2 = 40 \]
Theorem compile_correct :=
  forall e,
  [eval e] = execute (compile e []).

QuickChick compile_correct.
Theorem compile_correct :=
  forall e,
  [eval e] = execute (compile e [])?.

QuickChick compile_correct.
A look under the hood

**QuickChick compile_correct.**

- Extract “compile_correct” to a single ML file
- Compile (w/ optimizations)
- Run binary
Batch Execution – the command line tool

(*! QuickChick compile_correct. *)

• No overhead when compiling theories
• Use external tool to run all tests with one extraction/compilation!
Mutation testing

How do you know when you’re done?

• ... no bugs exist?
• ... testing is not good enough?
Mutation testing

How do you know when you’re done?

- ... no bugs exist?
- ... testing is not good enough?

```
Fixpoint compile (e : exp) :=
... (*! *) (*!! MINUS_ASSOC *)
  | AMinus e1 e2 => compile e2 ++ compile e1 ++ [SMinus]
(*!
  | AMinus e1 e2 => compile e1 ++ compile e2 ++ [SMinus]
*) ...
```
The road so far...

∀x.p(x)
The road so far...

\[ \forall x. p(x) \]

\[ \forall x. \ p(x) \rightarrow q(x) \]
The road so far...

\[ \forall x. \, p(x) \]

\[ \forall x. \, p(x) \to q(x) \]

If $x$ is well typed
The road so far...

∀x. p(x)

∀x. p(x) → q(x)

If x is well typed

Then it is either a value or can take a step
Properties with preconditions

∀x. p(x) → q(x)
Properties with preconditions

∀x. p(x) → q(x)
Properties with preconditions

\[ \forall x. p(x) \rightarrow q(x) \]
Properties with preconditions

\[ \forall x. \ p(x) \rightarrow q(x) \]

- Generate \( x \)
- Check \( p(x) \)
- If check succeeds, test \( q(x) \)
Properties with preconditions

\[ \forall x. p(x) \rightarrow q(x) \]

- Generate \( x \)
- Check \( p(x) \)
- If check succeeds, test \( q(x) \)
- If not, start over
Properties with preconditions

\[ \forall x. \ p(x) \rightarrow q(x) \]

Generate \( x \)

Check \( p(x) \)

If not, start over

If check succeeds, test \( q(x) \)
Let’s generate well-typed terms!

**GOAL:** Generate e, such that \((\text{well\_typed } e \ T)\) holds for a given T
Take 1 – Generate and test

• Assume we can *decide* whether a term has a given type
• Generate random lambda terms
• Filter out the ill-typed ones
Take 2 – Custom generators

Solution: Write a generator that produces well-typed terms!
The road ahead...

• Generators
  • Generation Monad
  • Primitive Generators
  • Generator Combinators – Trees!

• Properties
  • Decidability
  • The Checkable Class
  • An example: Mirrored Trees

• Open Research!
Class Gen (A : Type) := \{ arbitrary : G A \}.
Class Gen (A : Type) := \{ \text{arbitrary} : G A \}.
The G Monad

\[
\text{returnGen} : A \rightarrow G A.
\]

\[
\text{bindGen} : G A \rightarrow (A \rightarrow G B) \rightarrow G B.
\]
The G Monad

returnGen : A \rightarrow G A.

Always return the input argument, no randomness

bindGen : G A \rightarrow (A \rightarrow G B) \rightarrow G B.
The G Monad

\[
\text{returnGen} : \ A \rightarrow \ G \ A.
\]

- Always return the input argument, no randomness

\[
\text{bindGen} : \ G \ A \rightarrow (A \rightarrow G \ B) \rightarrow G \ B.
\]

- And a way to generate B given A...

- Given generator for A...
The G Monad

\[
\text{returnGen} : \ A \to \ G\ A.
\]

Always return the input argument, no randomness

\[
\text{bindGen} : \ G\ A \to (A \to \ G\ B) \to \ G\ B.
\]

And a way to generate B given A...

Given generator for A...

Produce a generator for B
choose : Choosable A -> A * A -> G A

randomR : A * A -> RandomSeed -> A * RandomSeed
Primitive Generators - lists

**listOf** : $G \ A \rightarrow G(\text{list } A)$

**vectorOf** : $\text{nat} \rightarrow G\ A \rightarrow G(\text{list } A)$

Size of list to be generated
Primitive Generators - lists

**listOf**: $G \ A \rightarrow G \ (\text{list} \ A)$

How does $\text{listOf}$ decide the size of the generated list?

**vectorOf**: $\text{nat} \rightarrow G \ A \rightarrow G \ (\text{list} \ A)$

Size of list to be generated
The G Monad – revisited

Inductive G A :=
| MkG : (nat -> RandomSeed -> A) -> G A.
Inductive Tree A :=
| Leaf : Tree A
| Node : A -> Tree A -> Tree A -> Tree A.
Generator Combinator - oneOf

oneOf_ : G A -> list (G A) -> G A
Generator Combinator - oneOf

```
oneOf_ : G A -> list (G A) -> G A

Picks one at random
```
Generator Combinator - oneOf

oneOf_ : G A -> list (G A) -> G A

Default element

Picks one at random
Generator Combinator - oneOf

OneOf_ : G A -> list (G A) -> G A

Default element

Notation oneOf : list (G A) -> G A

Picks one at random
A (naïve) random generator for trees

Fixpoint genTree: G tree :=
    oneOf [ ret Leaf
             , x <- arbitrary;
             l <- genTree;
             r <- genTree;
             ret (Node x l r) ].
A (naïve) random generator for trees

Fixpoint genTree : G tree :=
  oneOf [ ret Leaf
    , x <- arbitrary;
    l <- genTree;
    r <- genTree;
    ret (Node x l r) ].

The type of tree generators
A (naïve) random generator for trees

Fixpoint genTree: G tree :=
    oneOf [ ret Leaf
          , x <- arbitrary;
          l <- genTree;
          r <- genTree;
          ret (Node x l r) ].
A (naïve) random generator for trees

Fixpoint genTree : G tree :=
  oneOf [ ret Leaf
    , x <- arbitrary;
    l <- genTree;
    r <- genTree;
    ret (Node x l r) ].
A (naïve) random generator for trees

Fixpoint genTree: G tree :=
  oneOf [ ret Leaf
    , x <- arbitrary;
    l <- genTree;
    r <- genTree;
    ret (Node x l r) ].

The type of tree generators

Point distribution \{\text{Leaf}\}

Uniform choice
A (naïve) random generator for trees

Fixpoint genTree: G tree :=
  oneOf [ ret Leaf,
    x <- arbitrary;
    l <- genTree;
    r <- genTree;
    ret (Node x l r) ].

The type of tree generators

Point distribution {Leaf}

Uniform choice

• Why does this terminate? (it doesn’t)
• Is the distribution useful? (low probability of interesting trees)
A (naïve) random generator for trees

Fixpoint genTree: G tree :=
  oneOf [ ret Leaf 
    , x <- arbitrary; 
    l <- genTree; 
    r <- genTree; 
    ret (Node x l r) ].

The type of tree generators

Point distribution {Leaf}

Uniform choice

• Why does this terminate? (it doesn’t)
• Is the distribution useful? (low probability of interesting trees)
A (naïve) random generator for trees

```ocaml
Fixpoint genTree : G tree :=
  oneOf [ ret Leaf, x <- arbitrary; l <- genTree; r <- genTree; ret (Node x l r) ].
```

The type of tree generators

- **Uniform choice**
- **Point distribution** \{Leaf\}

- Why does this terminate? (it doesn’t)
- Is the distribution useful? (low probability of interesting trees)
A (naïve) random generator for trees

```ocaml
Fixpoint genTree : G tree :=
  oneOf [ ret Leaf,
          x <- arbitrary;
          l <- genTree;
          r <- genTree;
          ret (Node x l r) ].
```

The type of tree generators

- $x \in \mathbb{N}$
- $l \in \text{Tree}$
- $r \in \text{Tree}$

Point distribution

- \{Leaf\}

Uniform choice

- Why does this terminate? (it doesn’t)
- Is the distribution useful? (low probability of interesting trees)
A (better) random generator for trees

Fixpoint genTree (size : nat) : G tree :=

size parameter : upper limit of the depth of the tree

\{ t | size(t) \leq size \}
A (better) random generator for trees

\[
\text{Fixpoint } \text{genTree} \ (\text{size} : \text{nat}) : \text{G tree} := \\
\text{match size with} \\
| 0 => \text{ret Leaf} \\
| S \text{ size'} => \\
\{ t \mid \text{size}(t) \leq \text{size} \}
\]
A (better) random generator for trees

Fixpoint genTree (size : nat) : G tree :=
match size with
| O => ret Leaf
| S size’ =>
  oneOf [ ret Leaf
         , x <- arbitrary;
         l <- genTree size’;
         r <- genTree size’;
         ret (Node x l r) ].

size parameter : upper limit of the depth of the tree

if size = 0

if size = size’ + 1

Recursive calls with smaller size

\{ t | size(t) ≤ size \}
Generator Combinator - frequency

\[
\text{freq}_\_ : \text{G A} \rightarrow \text{list (nat} \times \text{G A}) \rightarrow \text{G A}
\]

- **Default element**
- **Picks one at random, using the weights!**

**Notation** freq : list (nat \times G A) \rightarrow G A
A (better) random generator for trees

Fixpoint genTree (size : nat) : G tree :=
match size with
| O => ret Leaf
| S size' =>
  oneOf [ ret Leaf,
    x <- arbitrary;
    l <- genTree size';
    r <- genTree size';
    ret (Node x l r) ].

size parameter : upper limit of the depth of the tree
\{ t | size(t) \leq size \}
A (better) random generator for trees

Fixpoint genTree (size : nat) : G tree :=
match size with
| O => ret Leaf
| S size' =>
  freq [ (1, ret Leaf)
  , (size, x <- arbitrary;
     l <- genTree size';
     r <- genTree size';
     ret (Node x l r)) ].

size parameter: upper limit of the depth of the tree

\{ t | \text{size}(t) \leq \text{size} \}
Let’s talk properties…

• Results
• Decidability
• Property combinators
• An example: tree mirroring
Inductive Result := Success | Failure.
Inductive Result := Success | Failure.

Definition Checker := G Result.

A Checker is just a randomized test that succeeds or fails.
Inductive Result := Success | Failure.

Definition Checker := G Result.

Class Checkable A :=
  { checker : A -> Checker }.

A Checker is just a randomized test that succeeds or fails.
Booleans are checkable

Instance CheckableBool : Checkable bool :=
{ checker b :=
    if b then ret Success else ret Failure
    }

Decidable Properties are Checkable

Class Dec P := \{ \text{dec} : \{P\} + \{\neg P\} \}.
Notation P? := \text{if dec } P \text{ then true else false.}
Decidable Properties are Checkable

Class Dec P := \{ \text{dec} : \{P\} + \{\sim P\} \}.
Notation P? := \text{if dec P then true else false}.

Instance CheckDec \{P\} \{Dec P\} : Checkable P :=
\{ \text{checker p := checker (P?)} \}.
Fixpoint mirror t : tree :=
  match t with
  | Leaf => Leaf
  | Node x l r => Node x (mirror r) (mirror l)
end.
Fixpoint mirror t : tree :=
  match t with
  | Leaf => Leaf
  | Node x l r => Node x (mirror r) (mirror l)
end.

Definition mirrorP (t : tree nat) :=
mirror (mirror t) = t.
Instance CheckFun {A P} {Gen A} {Checkable P} : Checkable (A -> P) :=
{
  checker f :=
    a <- arbitrary ;
    checker (f a)
}.
Class Shrink A := { shrink : A -> list A }.
Class Shrink A := { shrink : A \rightarrow list A }.
Class Shrink A := { shrink : A → list A }.
Class Shrink A := \{ shrink : A \rightarrow \text{list A} \}.
Class Shrink $A := \{ \text{shrink} : A \rightarrow \text{list} A \}$. 

Original Counterexample $x$

- $X_1$
- $X_2$
- $X_3$
Class Shrink $A := \{ \text{shrink} : A \to \text{list } A \}$. 

Original Counterexample $x$

- $X1$
- $X2$
- $X3$

- $X21$
- $X22$
- $X23$
∀x. p(x) → q(x)
Back to Preconditions!

\[ \forall x. \ p(x) \rightarrow q(x) \]

\begin{verbatim}
insert : A -> list A -> list A
sorted : list A -> bool

Definition insert_spec_sorted :=
    forall x l,
        sorted l -> sorted (insert x l).
\end{verbatim}
Problem: Writing a good Generator
Problem: Writing a good Generator

All generated lists are sorted
Problem: Writing a good Generator

All generated lists are sorted

All sorted lists can be generated
Problem: Writing a good Generator

- All generated lists are sorted
- All sorted lists can be generated
- Distribution appropriate for testing
Take 2 – Custom Generators

Solution: Write a generator that produces well-typed terms!

Problem: Writing a good generator
Problem: Too much boilerplate
Problem: Maintenance nightmare!

Testing feedback should be immediate

Can be very complex [ICFP 2013]

Generators and predicates must be kept in sync
Solution – Derive Generators Automatically

IDEA: Given a predicate p, produce a generator automatically

• Constrained Data with Uniform Distributions [FLOPS 2014]
• Making Random Judgments [ESOP 2015]
• Luck [POPL 2017]
• Generating Good Generators [POPL 2018]
Solution – Derive Generators Automatically

IDEA: Given a predicate p, produce a generator automatically

• Constrained Data with Uniform Distributions [FLOPS 2014]
• Making Random Judgments [ESOP 2015]
• Luck [POPL 2017]
• Generating Good Generators [POPL 2018]

Problems:
• Addressing larger classes of preconditions
• Efficiency/Optimization
• Correctness
• Distribution <- big one!
More Open Problems!

- Deriving decidability procedures for inductive relations
- Shrinking, formalization and automation
- Connections with other areas
  - Probabilistic Programming
  - Fuzz Testing
  - Machine Learning
Thank you!