QuickChick : Property-Based Testing in Coq

POPL 2019

Tutorial Fest

14/01/2019

Leonidas Lampropoulos

Benjamin C. Pierce

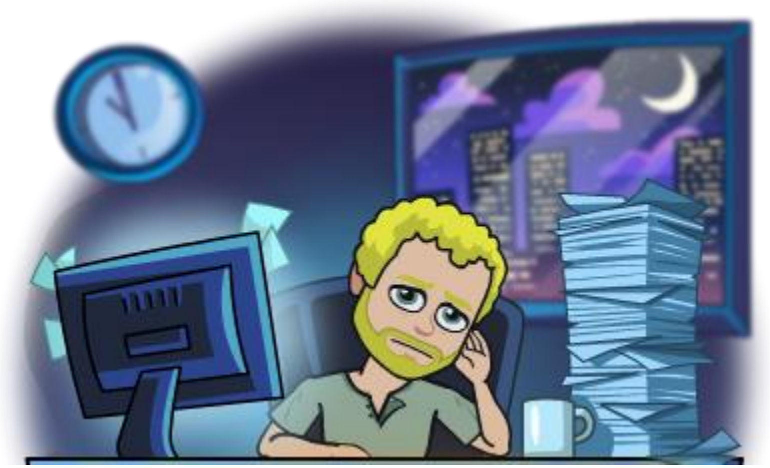
Software Foundations Volume

High-level view of workflow

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

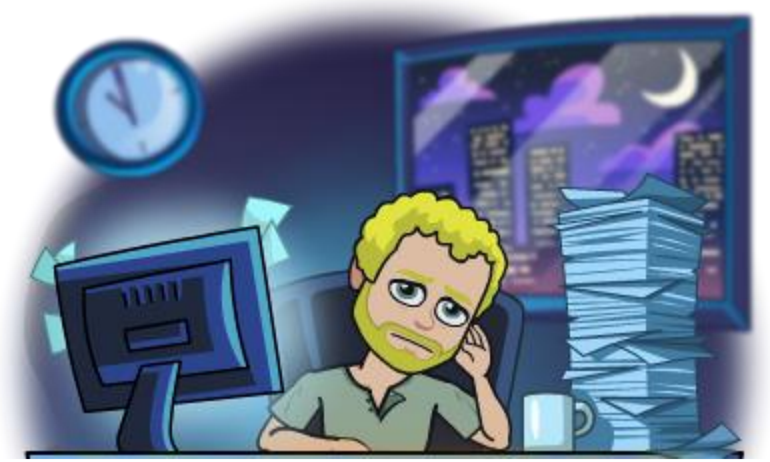
High-level view of workflow

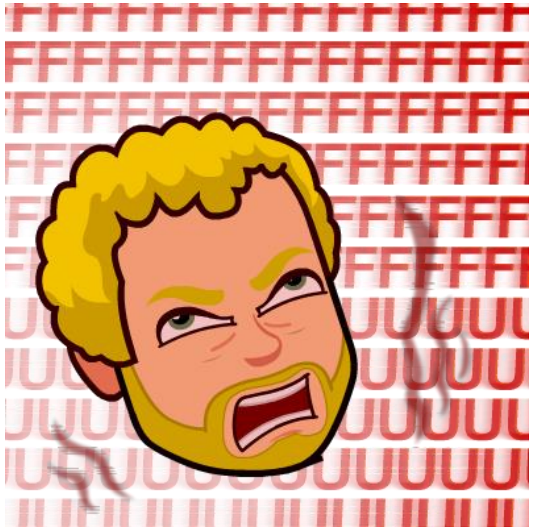
```
Theorem preservation :=
  forall e T Γ,
     Γ |- e : T -> e => e’ -> Γ |- e’ : T.
```



High-level view of workflow

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





A better workflow

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

A better workflow

```
Theorem preservation :=
  forall e τ Γ,
     Γ |- e : τ -> e => e’ -> Γ |- e’ : τ.
```

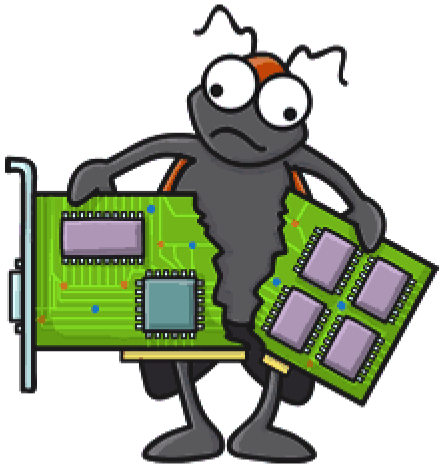
A better workflow

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

A better workflow

```
Theorem preservation :=
  forall 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

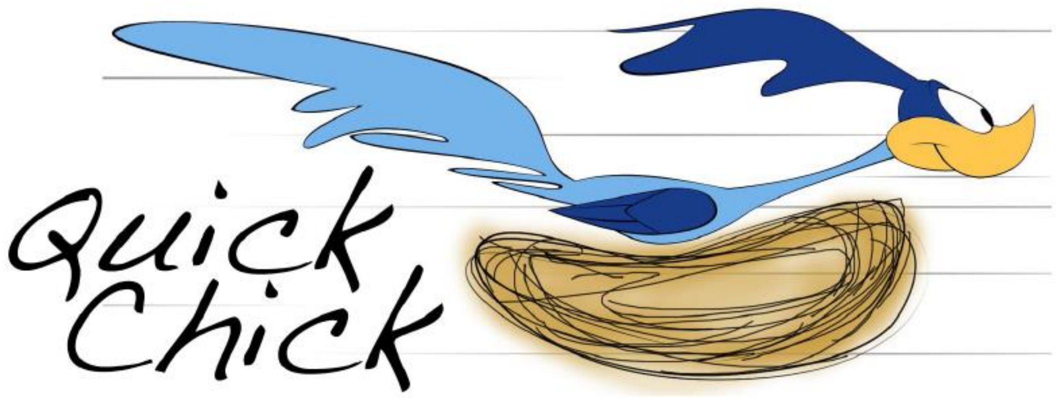
- Strong formal guarantees
- Recent success stories
  - [CompCert] : Verified optimizing C compiler
  - [CertiKOS] : Extensible architecture for certified OS kernels
- …but still very expensive

Testing and Verification

- 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 T Γ,
     Γ |- e : T -> e => e’ -> Γ |- e’ : T.
```

```
QuickChick preservation.
```

Overview of property-based testing

Decidable Property

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

```
QuickChick preservation.
```

Overview of property-based testing

Generate random inputs

Decidable Property

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

```
QuickChick preservation.
```

Overview of property-based testing

Generate random inputs

Decidable Property

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

```
QuickChick preservation.
```

Test result

Overview of property-based testing

Generate random inputs

Decidable Property

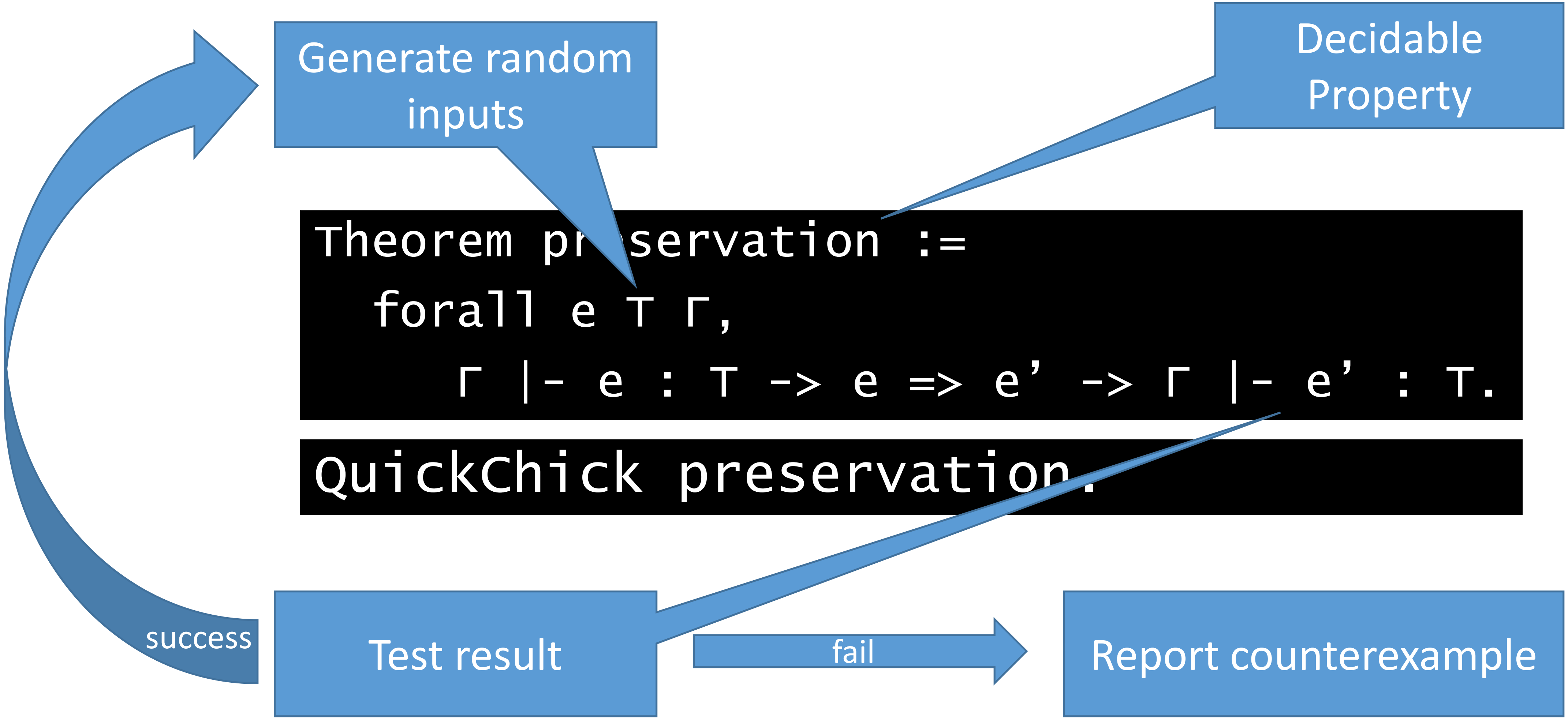
```
Theorem preservation :=
  forall e T Γ,
     Γ |- e : T -> e => e’ -> Γ |- e’ : T.
```

```
QuickChick preservation.
```

success

Test result

Overview of property-based testing



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	`ANum 42`
17 + 25	`APlus (ANum 17) (ANum 25)`
(22 - 1) * 2	`AMult (AMinus (ANum 22) (ANum 1)) (ANum 2)`

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

Arithmetic Expressions - Printing

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

Arithmetic Expressions - Printing

```
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.
```

Arithmetic Expressions - Printing

Explicitly remember how to print nats

```
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.
```

The Show TypeClass

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

The Show TypeClass

Class method

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

The Show TypeClass

Class method

```
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 ++ “)”
   | …
 |}.
```

The Show TypeClass

Class method

```
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

The Show TypeClass

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

Derive Show for exp.
```

Arithmetic Expressions - Generation

```
Class Gen (A : Type) := { arbitrary : G A }.

Derive Arbitrary for exp.
```

Arithmetic Expressions - Gene

Class method

```
Class Gen (A : Type) := { arbitrary : G A }.


Derive Arbitrary for exp.
```

Arithmetic Expressions - Gene

Class method

```
Class Gen (A : Type) := { arbitrary : G A }.

Derive Arbitrary for exp.
```

Generation Monad

Demo Time! - Sampling

Evaluation

```
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.
```

An expression optimizer

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 → x
- X + 0 → x
- X * 0 → 0
- X * 1 → x
- X – 0 → x
- …

$$\boxed{e : exp} \xrightarrow{\text{eval}} \boxed{n}$$

An expression optimizer - Correctness

Optimizations:

0 + x → x

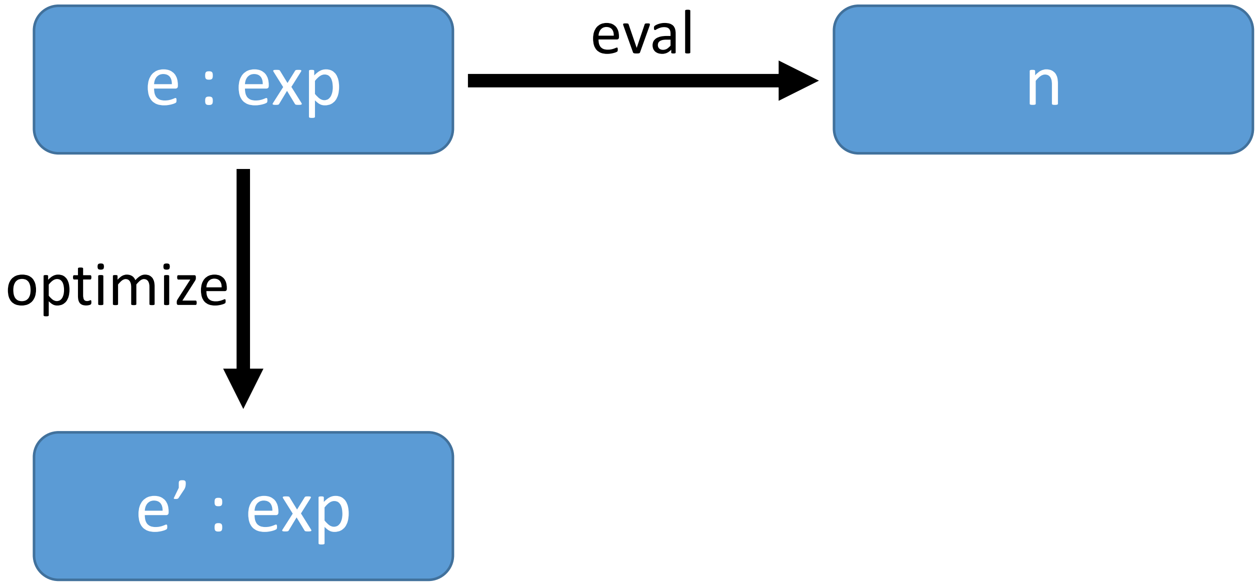
X + 0 → x

X * 0 → 0

X * 1 → x

X – 0 → x

…



An expression optimizer - Correctness

Optimizations:

0 + x → x

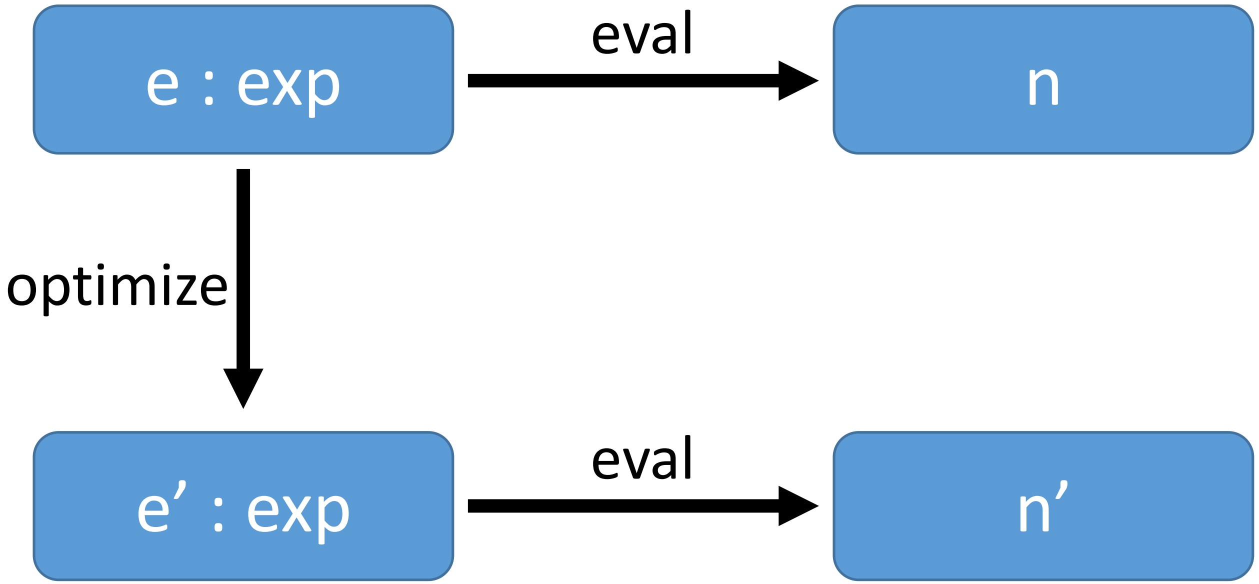
X + 0 → x

X * 0 → 0

X * 1 → x

X – 0 → x

…



An expression optimizer - Correctness

Optimizations:

0 + x → x

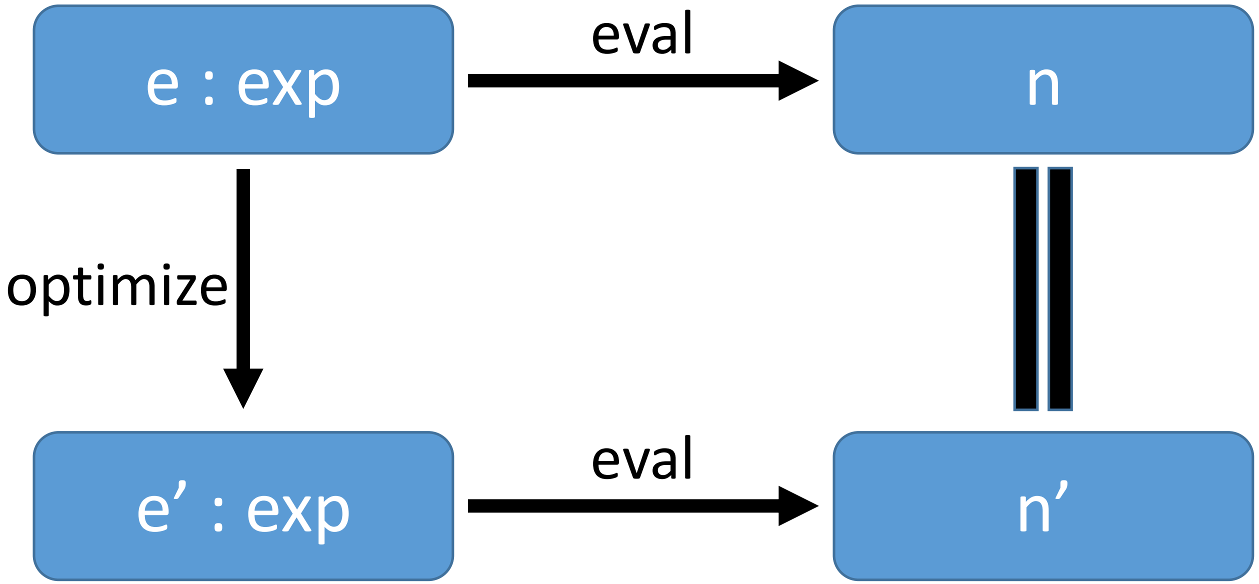
X + 0 → x

X * 0 → 0

X * 1 → x

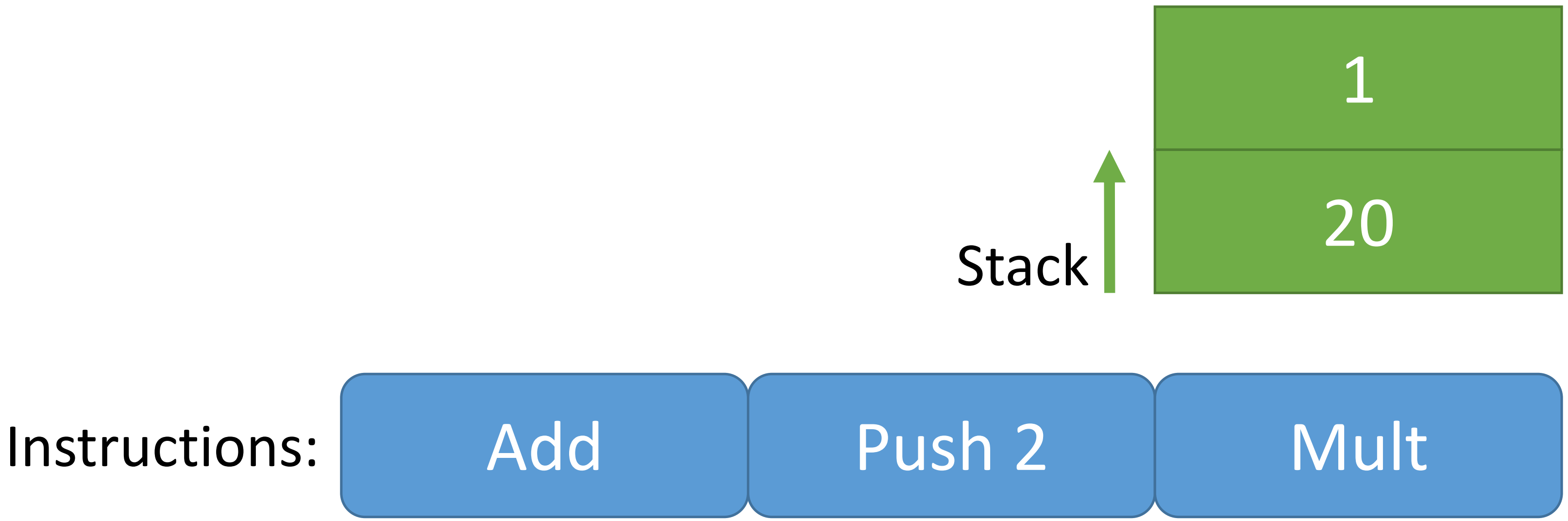
X – 0 → x

…

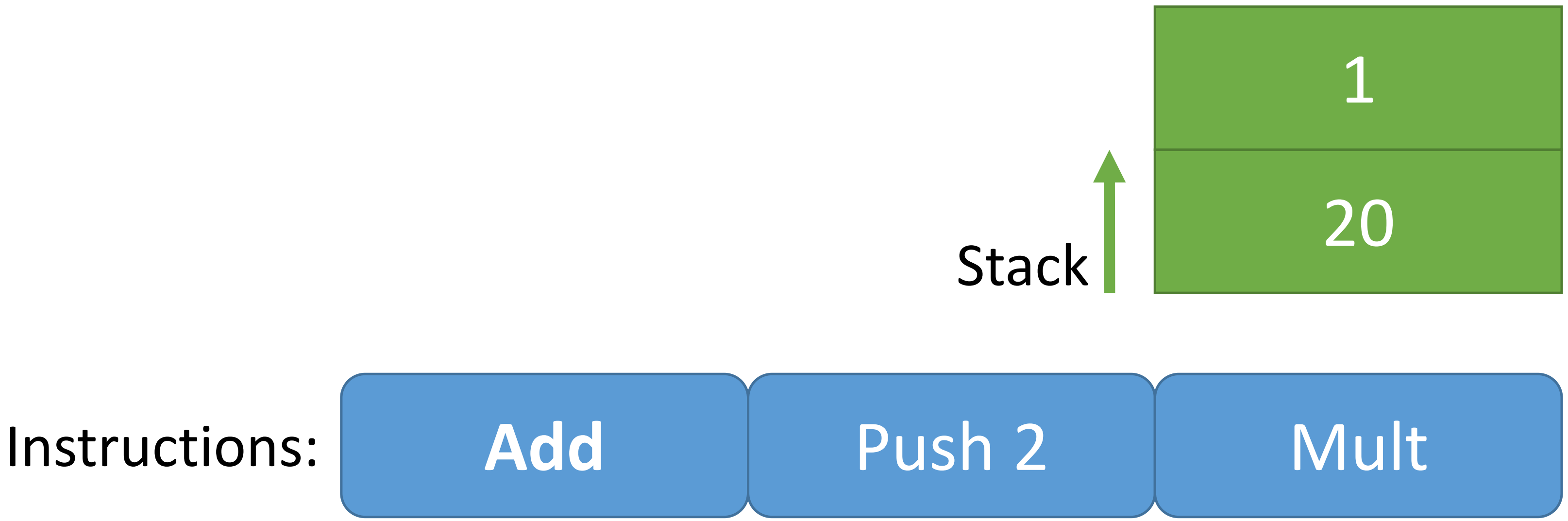


The Stack Machine

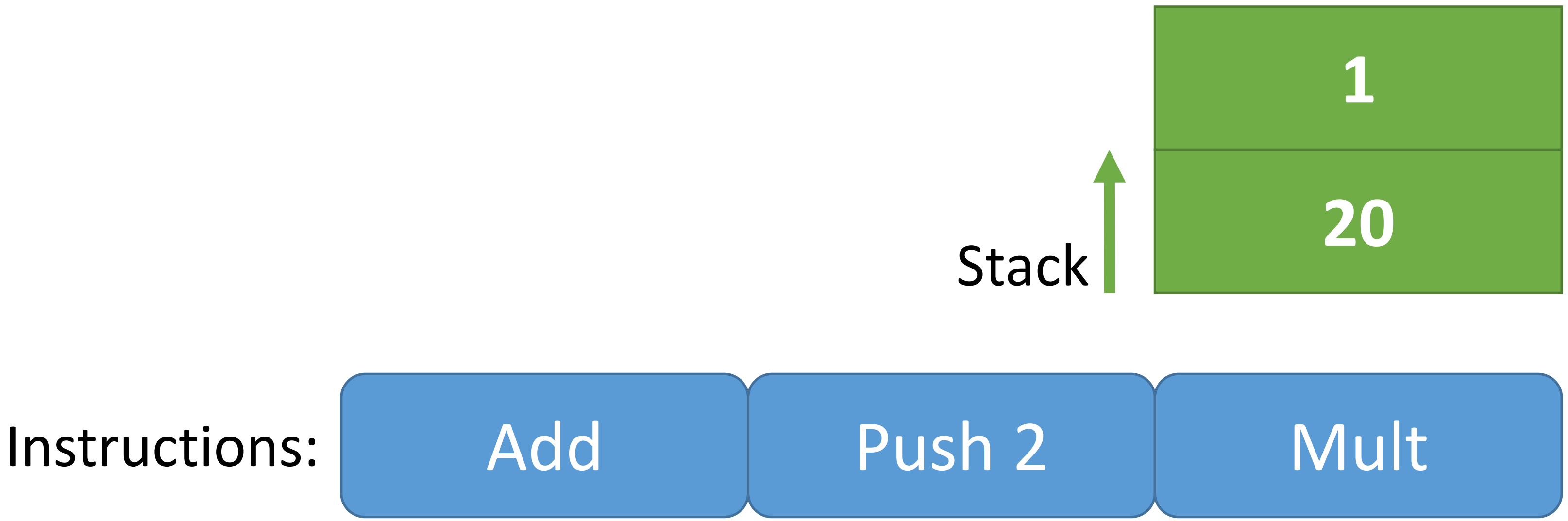
The Stack Machine



The Stack Machine



The Stack Machine



The Stack Machine

Stack

21

Instructions:

Push 2

Mult

The Stack Machine

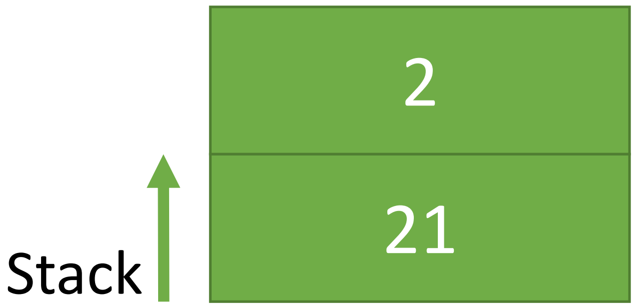
Stack

21

Instructions:

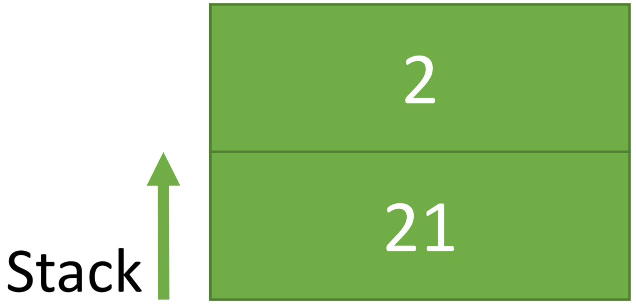
Push 2	Mult

The Stack Machine



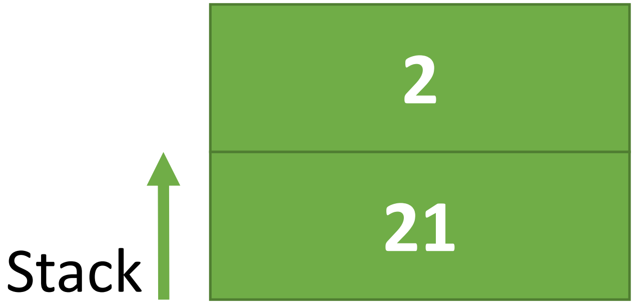
Instructions: Mult

The Stack Machine



Instructions: **Mult**

The Stack Machine



Instructions: Mult

The Stack Machine

Stack

42

Instructions:

The Compiler

(20 + 1) * 2

The Compiler

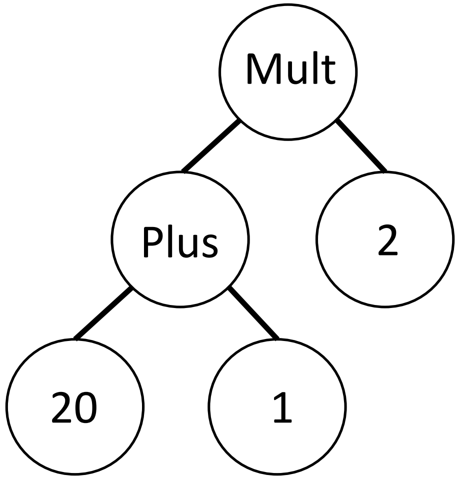
(20 + 1) * 2

```
AMult (APlus (Anum 20) (ANum 1)) (ANum 2)
```

The Compiler

(20 + 1) * 2

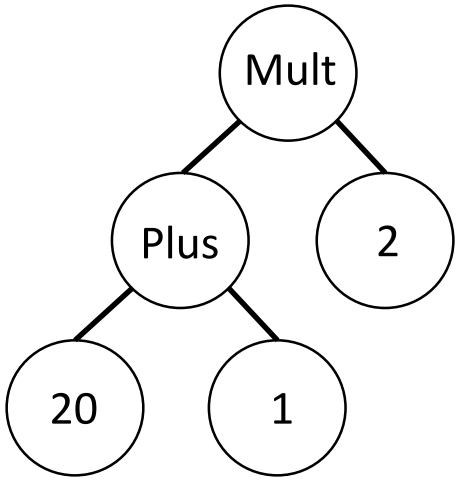
```
AMult (APlus (Anum 20) (ANum 1)) (ANum 2)
```



The Compiler

(20 + 1) * 2

```
AMult (APlus (Anum 20) (ANum 1)) (ANum 2)
```



Push 20	Push 1	Add	Push 2	Mult

The Compiler - Correctness

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

```
QuickChick compile_correct.
```

The Compiler - Correctness

Dec Typeclass

```
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…

$$\forall x . p(x)$$

The road so far…

$$\forall x. p(x)$$

$$\forall x.\ p(x) \to q(x)$$

The road so far…

$$\forall x. p(x)$$

$$\forall x.\ p(x) \rightarrow q(x)$$

If *x* is well typed

The road so far…

$$\forall x . p(x)$$

$$\forall x.\ p(x) \rightarrow q(x)$$

If *x* is well typed

Then it is either a value or can take a step

Properties with preconditions

$$\forall x.\ p(x) \rightarrow q(x)$$

Properties with preconditions

Generate x

$$\forall x.\ p(x) \to q(x)$$

Properties with preconditions

Generate x

$$\forall x.\ p(x) \rightarrow q(x)$$

Check $p(x)$

Properties with preconditions

Generate x

$$\forall x.\ p(x) \rightarrow q(x)$$

Check $p(x)$

If check succeeds, test $q(x)$

Properties with preconditions

Generate x

If not, start over

$$\forall x.\ p(x) \rightarrow q(x)$$

Check $p(x)$

If check succeeds, test $q(x)$

Properties with preconditions

Generate x

If not, start over

$$\forall x.\ p(x) \rightarrow q(x)$$

Check $p(x)$

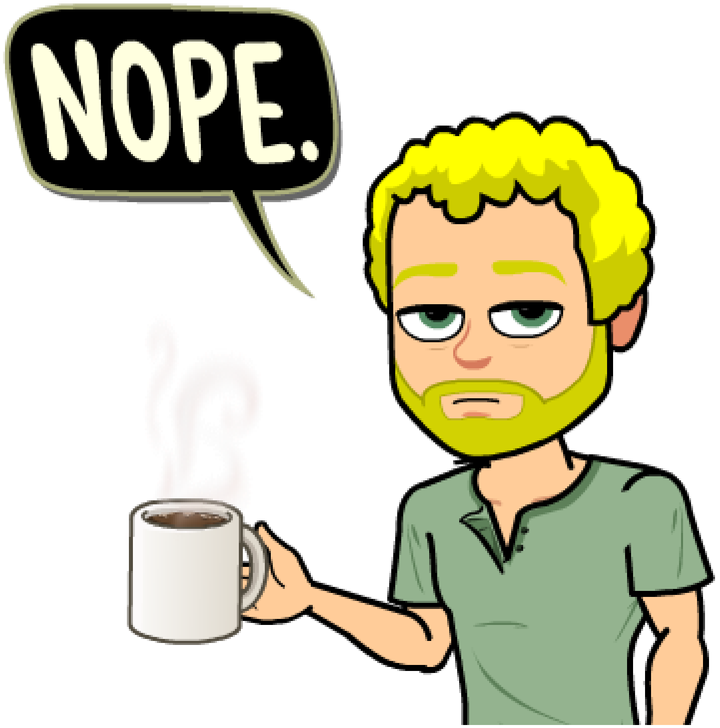
If check succeeds, test $q(x)$

Let’s generate well-typed terms!

GOAL: Generate e, such that (`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!

Generation Overview

```
Class Gen (A : Type) := { arbitrary : G A }.
```

Generation Overview

```
Class Gen (A : Type) := { arbitrary : G A }.
```

Functions from
random seeds to A

The G Monad

```
returnGen : A -> G A.
```

```
bindGen : G A -> (A -> G B) -> G B.
```

The G Monad

```
returnGen : A -> G A.
```

Always return the input argument, no randomness

```
bindGen : G A -> (A -> G B) -> G B.
```

The G Monad

```
returnGen : A -> G A.
```

Always return the input argument, no randomness

And a way to generate B given A…

```
bindGen : G A -> (A -> G B) -> G B.
```

Given generator for A…

The G Monad

```
returnGen : A -> G A.
```

Always return the input argument, no randomness

```
bindGen : G A -> (A -> G B) -> G B.
```

Given generator for A…

And a way to generate B given A…

Produce a generator for B

Primitive Generators - choose

```
choose : Choosable A -> A * A -> G A
```

Typeclass with randomR

```
randomR : A * A -> RandomSeed -> A * RandomSeed
```

Primitive Generators - lists

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

```
vectorOf : nat -> G A -> G (list A)
```

Size of list to be generated

Primitive Generators - lists

How does listOf decide the size of the generated list?

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

```
vectorOf : nat -> G A -> G (list A)
```

Size of list to be generated

The G Monad – revisited

```
Inductive G A :=
| MkG : (nat -> RandomSeed -> A) -> G A.
```

Custom Datatypes – Trees!

```
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

Picks one at random

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

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

The type of tree generators

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

A (naïve) random generator

The type of tree generators

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

Uniform choice

A (naïve) random generator

```
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

A (naïve) random generator

```
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

$x \in Nat$

$l \in Tree$

$r \in Tree$

A (naïve) random generator

The type of tree generators

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

Point distribution {Leaf}

$x \in Nat$

$l \in Tree$

$r \in Tree$

Uniform choice

- Why does this terminate? (it doesn't)
- Is the distribution useful? (low probability of interesting trees)

A (naïve) random generator

```
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

$x \in Nat$

$l \in Tree$

$r \in Tree$

- Why does this terminate? (it doesn't)
- Is the distribution useful? (low probability of interesting trees)

Leaf

A (naïve) random generator

The type of tree generators

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

Point distribution {Leaf}

Uniform choice

$x \in Nat$

$l \in Tree$

$r \in Tree$

- Why does this terminate? (it doesn't)
- Is the distribution useful? (low probability of interesting trees)

Leaf

Leaf

A (naïve) random generator

The type of tree generators

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

Point distribution {Leaf}

$x \in Nat$

$l \in Tree$

$r \in Tree$

Uniform choice

- Why does this terminate? (it doesn't)
- Is the distribution useful? (low probability of interesting trees)

Leaf

Leaf

Node 2 Leaf (Node 0 (Node 13 (Node 4 Leaf (Node 7 Leaf Leaf)) (Node 0 …

A (better) random generator for t

size parameter :
upper limit of the
depth of the tree

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

$\{t \mid size(t) \leq size\}$

A (better) random generator for t

size parameter : upper limit of the depth of the tree

```
Fixpoint genTree (size : nat) : G tree :=
  match size with
  | O => ret Leaf
  | S size' =>
```

$\{t \mid size(t) \leq size\}$

if size = 0

if size = size' + 1

A (better) random generator for t

size parameter : upper limit of the depth of the tree

```
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) ].
```

$\{t \mid size(t) \leq size\}$

if size = 0

if size = size' + 1

$x \in Nat$

$l \in \{t \mid size(t) \leq size'\}$

$r \in \{t \mid size(t) \leq size'\}$

Recursive calls with smaller size

Generator Combinator - frequency

```
freq_ : G A -> list (nat * G A) -> G A
```

Default element

Picks one at random, using the weights!

```
Notation freq : list (nat * G A) -> G A
```

A (better) random generator for t

size parameter : upper limit of the depth of the tree

```
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) ].
```

$\{t \mid size(t) \leq size\}$

A (better) random generator for t

size parameter : upper limit of the depth of the tree

$\{t \mid size(t) \leq size\}$

```
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)) ].
```

$\frac{1}{size+1}$ of the time

$\frac{size}{size+1}$ of the time

Let’s talk properties…

- Results
- Decidability
- Property combinators
- An example: tree mirroring

Results and Checkers

```
Inductive Result := Success | Failure.
```

Results and Checkers

```
Inductive Result := Success | Failure.

Definition Checker := G Result.
```

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

Results and Checkers

```
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 := { dec : {P} + {~P} }.
Notation P? := if dec P then true else false.
```

Decidable Properties are Checkable

```
Class Dec P := { dec : {P} + {~P} }.
Notation P? := if dec P then true else false.

Instance CheckDec {P} {Dec P} : Checkable P :=
{ checker p := checker (P?) }.
```

Tree mirroring

```
Fixpoint mirror t : tree :=
  match t with
  | Leaf => Leaf
  | Node x l r => Node x (mirror r) (mirror l)
  end.
```

Tree mirroring

```
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.
```

Putting it all together

```
Instance CheckFun {A P} {Gen A} {Checkable P}
: Checkable (A -> P) :=
{
  checker f :=
    a <- arbitrary ;
    checker (f a)
}.
```

Shrinking

```
Class Shrink A := { shrink : A -> list A }.
```

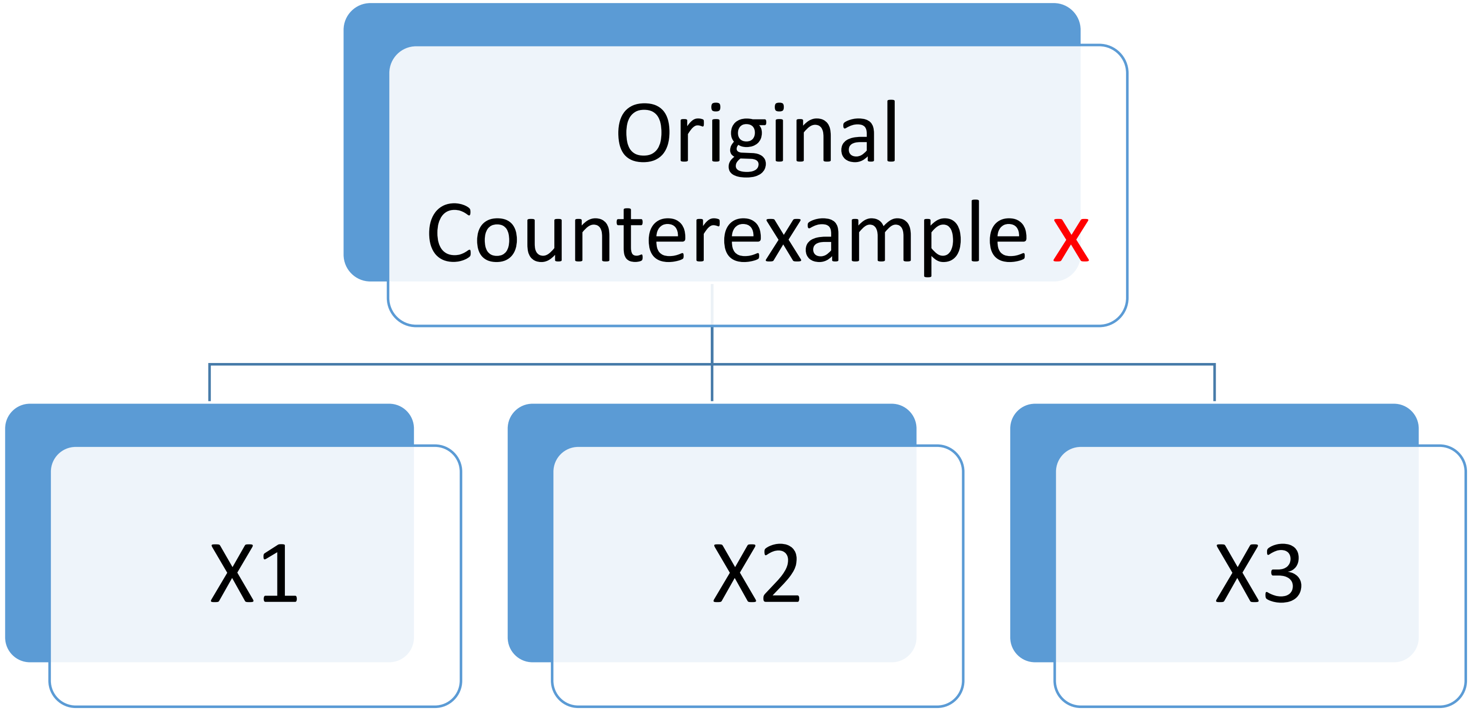
Shrinking

```
Class Shrink A := { shrink : A -> list A }.
```

Original Counterexample x

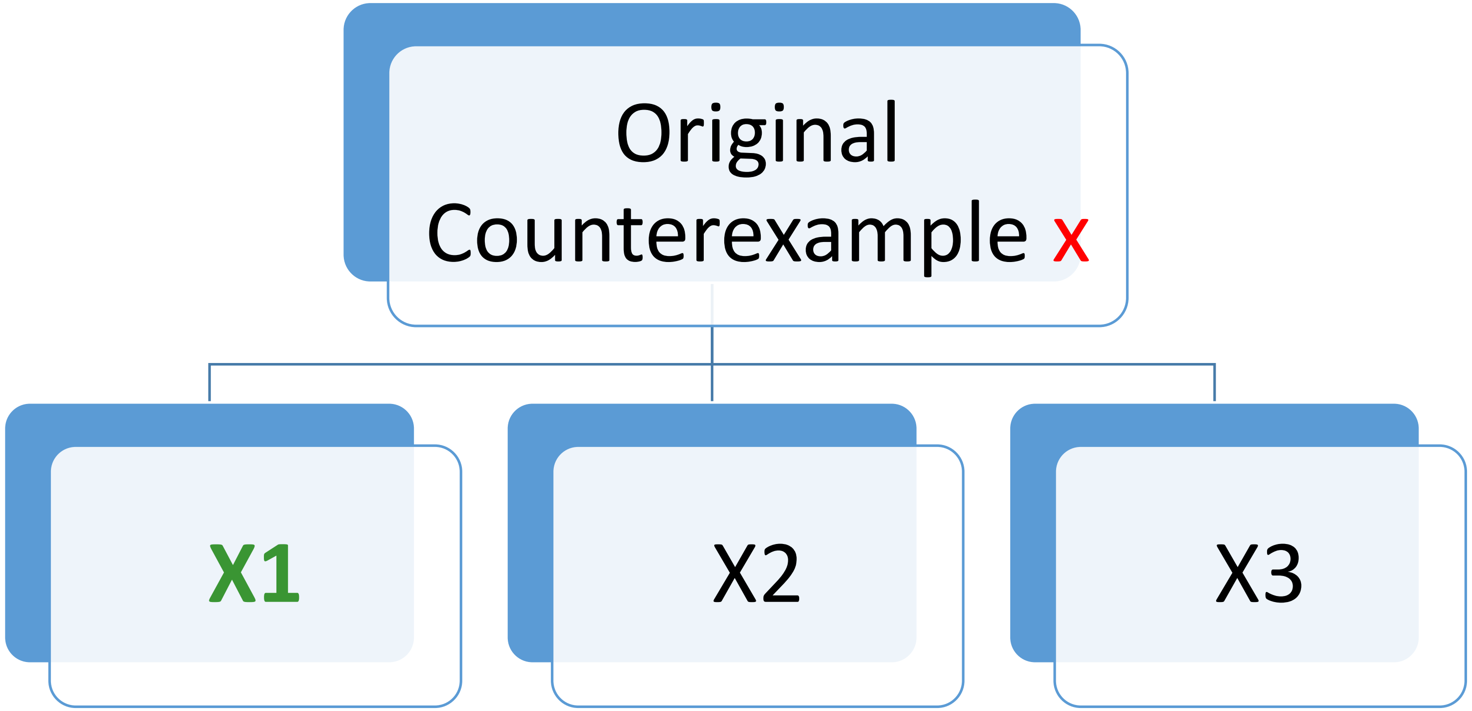
Shrinking

```
Class Shrink A := { shrink : A -> list A }.
```



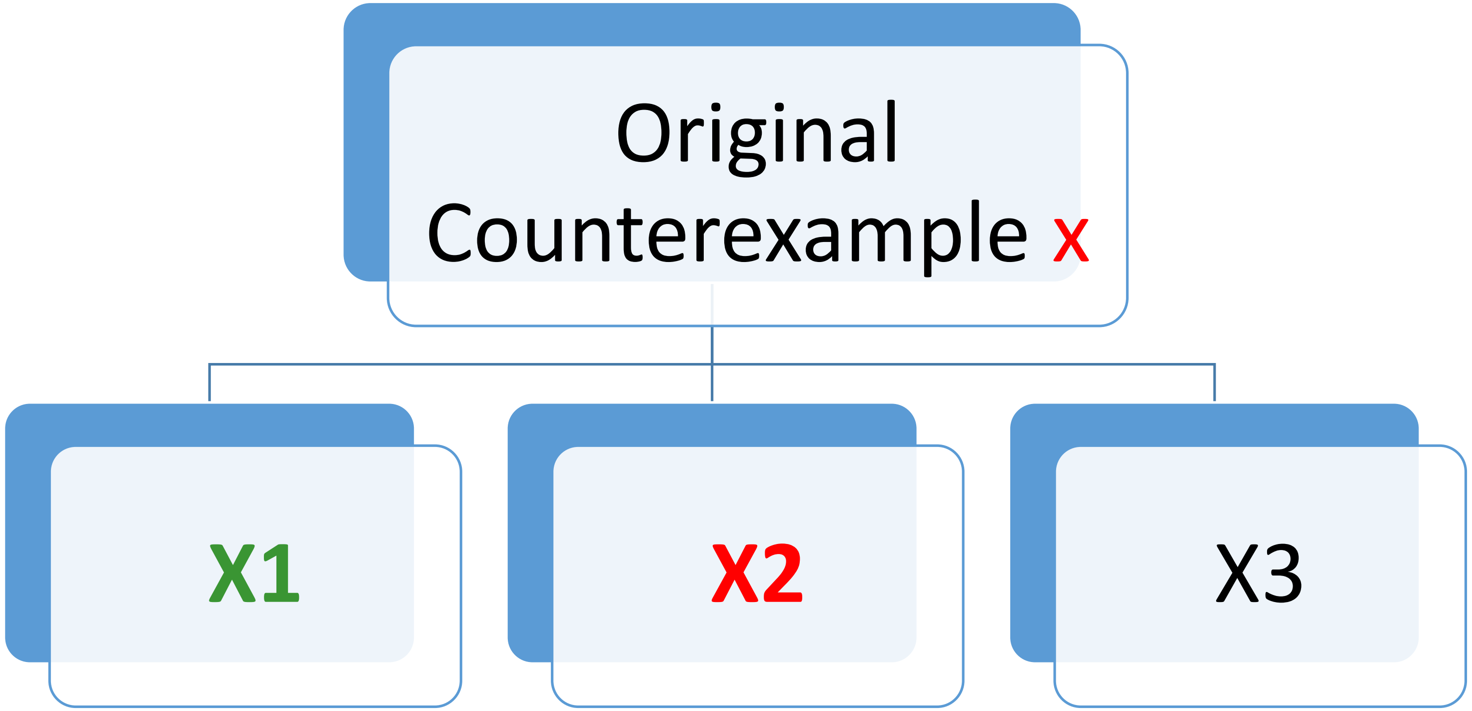
Shrinking

```
Class Shrink A := { shrink : A -> list A }.
```



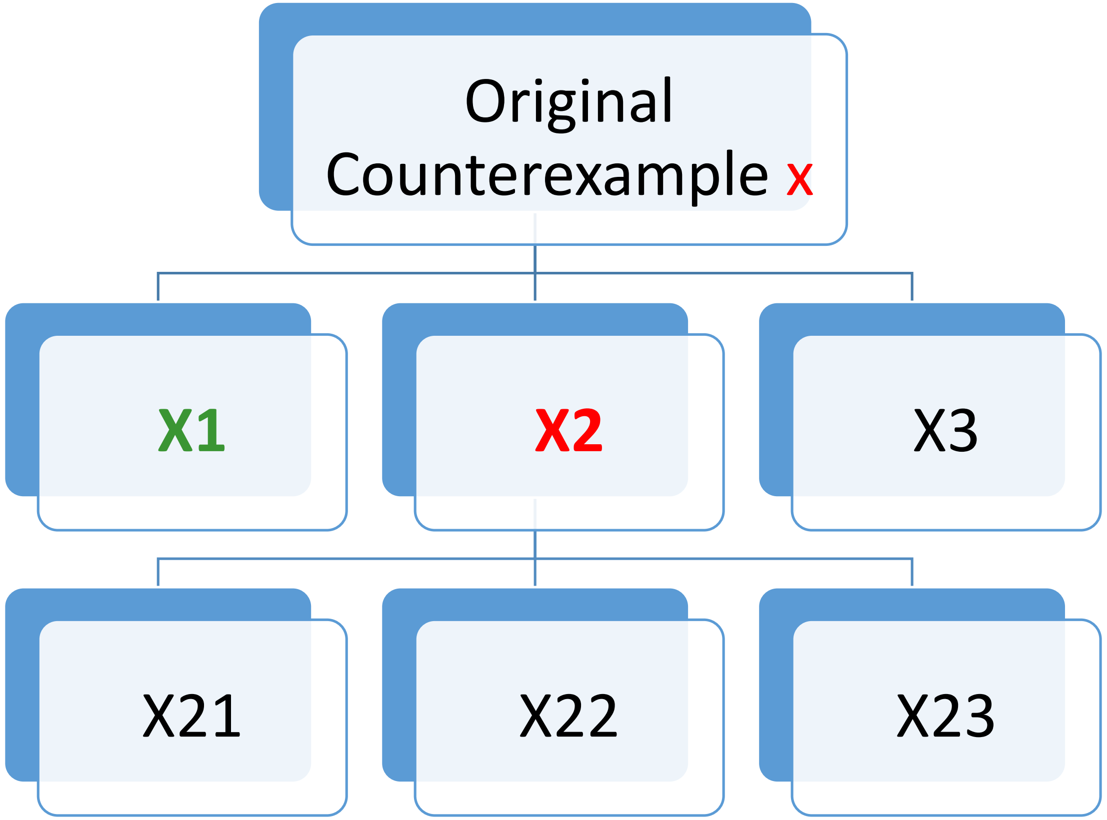
Shrinking

```
Class Shrink A := { shrink : A -> list A }.
```



Shrinking

```
Class Shrink A := { shrink : A -> list A }.
```



Back to Preconditions!

$$\forall x.\ p(x) \rightarrow q(x)$$

Back to Preconditions!

$$\forall x.\ p(x) \rightarrow q(x)$$

```
insert : A -> list A -> list A
sorted : list A -> bool

Definition insert_spec_sorted :=
  forall x l,
    sorted l -> sorted (insert x l).
```

Custom generators

Problem: Writing a good Generator

Custom generators

Problem: Writing a good Generator

All generated lists
are sorted

Custom generators

All sorted lists can be generated

Problem: Writing a good Generator

All generated lists are sorted

Custom generators

All sorted lists can be generated

Problem: Writing a good Generator

All generated lists are sorted

Distribution appropriate for testing

Take 2 – Custom Generators

Solution: Write a generator that produces well-type

Can be very complex
[ICFP 2013]

Problem: Writing a good generator

Problem: Too much boilerplate

Problem: Maintenance nightmare!

Testing feedback should be immediate

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!


SOFTWARE FOUNDATIONS
VOLUME 4
QuickChick: Property-Based Testing in Coq
Leonidas Lampropoulos
Benjamin C. Pierce