



# Chamelean

Property-Based Testing for Lean  
via Metaprogramming

Ernest Ng  
(advised by Cody Roux & Mike Hicks)

In collaboration with



# Property-Based Testing

## 1. Write *properties*

Spec for Binary Search Trees (BSTs):

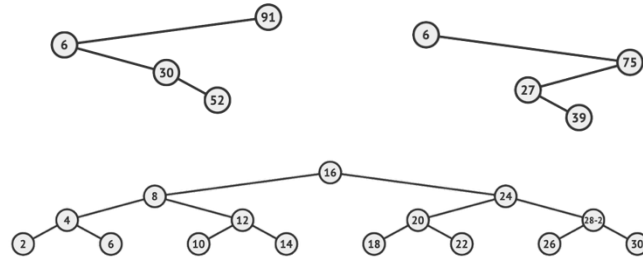
$$\forall x \text{ tree,} \\ \text{isBST tree} \\ \Rightarrow \text{isBST (insert x tree)}$$

# Property-Based Testing (PBT)

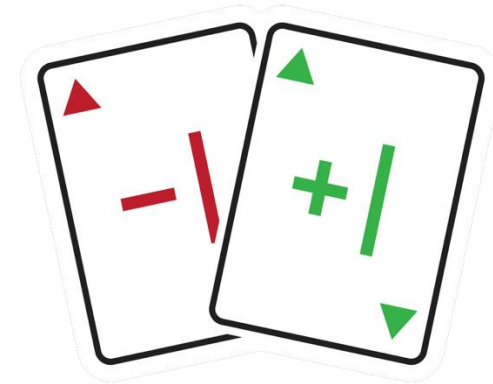
## 2. Generate *random inputs*



1, 2, 4, 7, 9, ...



## 3. Test if random inputs satisfy property

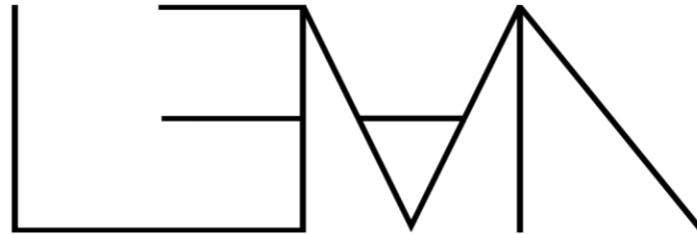


# Why should proof assistants support PBT?

Writing proofs is hard & time-consuming

⇒ Perform PBT before embarking on proofs!

# Every major proof assistant has a PBT framework



# Problem: writing PBT generators is hard

Cedar team: writing good generators took 6 months (estimated)



Jane Street devs: writing generators is **"tedious"** & **"high-effort"**

Goldstein et al. (ICSE '24)

# The Constrained Random Generation Problem

Given some proposition  $P$ ,  
*automatically* generate random values satisfying  $P$

# The Constrained Random Generation Problem

Given some *inductive relation*  $P$ ,  
*automatically* generate random values satisfying  $P$



# Inductive Relations

`isBST lo hi tree`

"is `tree` a `BST` containing values between `lo` & `hi`?"

`inductive isBST : Nat → Nat → Tree → Prop where`

`...`

# Inductive Relations

`isBST lo hi tree`

"is `tree` a `BST` containing values between `lo` & `hi`?"

```
inductive isBST : Nat → Nat → Tree → Prop where  
  | BSTLeaf : ∀ lo hi, isBST lo hi Leaf  
  ...
```

# Inductive Relations

`isBST lo hi tree`

"is `tree` a `BST` containing values between `lo` & `hi`?"

```
inductive isBST : Nat → Nat → Tree → Prop where
| BSTLeaf : ∀ lo hi, isBST lo hi Leaf
| BSTNode : ∀ lo hi x l r,
  lo < x < hi →
  isBST lo x l →
  isBST x hi r →
  isBST lo hi (Node x l r)
```

## isBST lo hi tree

"is tree a BST containing values between lo & hi?"

### Function

```
def isBST :  
  Nat → Nat → Tree → Bool :=  
  ...
```

- ✓ Can be executed!
- ✗ Can't encode all properties
- ✗ Hard to reason inductively

### Inductive Relation

```
inductive isBST :  
  Nat → Nat → Tree → Prop where  
  ...
```

- ✓ Facilitates inductive reasoning!
- ✓ Easy to model properties!
- ✗ No computational content  
(Chameleon addresses this issue!)

# Chamelean

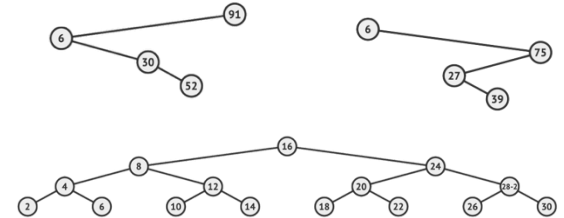
1. User specifies a Lean inductive relation

```
inductive isBST :
```

```
  Nat → Nat → Tree → Prop where
```

```
| BSTLeaf : ...
```

```
| BSTNode : ...
```



(a generator of **Tree**s satisfying **isBST**)

```
#derive_generator (fun tree => isBST lo hi tree)
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  ...
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' => ...
```

pattern-match on **fuel** parameter

(needed to make generators  
structurally decreasing)

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero ⇒ return Leaf  
  | succ fuel' ⇒ ...
```

input **t** unified with **Leaf**



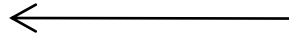
```
inductive isBST (lo : Nat) (hi : Nat) (t : Tree) : Prop where  
  | BSTLeaf : ∀ lo hi, isBST lo hi Leaf
```



# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => return Leaf  
  | succ fuel' =>  
    backtrack [  
      ...  
    ]
```

**backtrack** keeps picking from a (weighted)  
list of sub-generators until one succeeds



# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => return Leaf  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf), ← input t unified with Leaf  
      ...  
    ]
```

```
inductive isBST (lo : Nat) (hi : Nat) (t : Tree) : Prop where  
  | BSTLeaf : ∀ lo hi, isBST lo hi Leaf
```

# Anatomy of a Generator

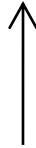
```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf),  
      (succ fuel', do  
        ...
```

(Recursive case)

```
| BSTNode :  $\forall$  lo hi x l r,  
  lo < x < hi  $\rightarrow$   
  isBST lo x l  $\rightarrow$   
  isBST x hi r  $\rightarrow$   
  isBST lo hi (Node x l r)
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf),  
      (succ fuel', do  
        let x ← genSuchThat (fun x => lo < x < hi)  
        ...
```



Type class method for the derived generator  
associated with `lo < x < hi`

```
| BSTNode : ∀ lo hi x l r,  
  lo < x < hi →  
  ...
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf),  
      (succ fuel', do  
        let x ← genSuchThat (fun x => lo < x < hi)  
        let l ← genBST lo x fuel'  
        ...  
      )  
    ]
```

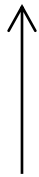


Recursively generate left subtree **l**

```
| BSTNode : ∀ lo hi x l r,  
  lo < x < hi →  
  isBST lo x l →  
  ...
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf),  
      (succ fuel', do  
        let x ← genSuchThat (fun x => lo < x < hi)  
        let l ← genBST lo x fuel'  
        let r ← genBST x hi fuel'  
        ...  
      )  
    ]
```



Recursively generate right subtree *r*

```
| BSTNode : ∀ lo hi x l r,  
  lo < x < hi →  
  isBST lo x l →  
  isBST x hi r →  
  ...
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf),  
      (succ fuel', do  
        let x ← genSuchThat (fun x => lo < x < hi)  
        let l ← genBST lo x fuel'  
        let r ← genBST x hi fuel'  
        return (Node x l r))  
    ]
```

Return a **Node**



```
| BSTNode : ∀ lo hi x l r,  
  lo < x < hi →  
  isBST lo x l →  
  isBST x hi r →  
  isBST lo hi (Node x l r)
```

# Anatomy of a Generator

```
def genBST (lo : Nat) (hi : Nat) (fuel : Nat) : Gen Tree :=  
  match fuel with  
  | zero => ...  
  | succ fuel' =>  
    backtrack [  
      (1, return Leaf),  
      (succ fuel', do  
        let x ← genSuchThat (fun x => lo < x < hi)  
        let l ← genBST lo x fuel'  
        let r ← genBST x hi fuel'  
        return (Node x l r)  
      ]
```

inductive isBST

(lo : Nat)

(hi : Nat)

(t : Tree) : Prop where

| BSTNode :  $\forall$  lo hi x l r,

lo < x < hi  $\rightarrow$

isBST lo x l  $\rightarrow$

isBST x hi r  $\rightarrow$

isBST lo hi (Node x l r)

input **t** unified with (Node x l r)

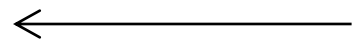




# Some Generators are Better Than Others

A *naïve* BST generator

`let x ← arbitrary`



Generate some `arbitrary` (unconstrained) `x`

# Some Generators are Better Than Others

A *naïve* BST generator

```
let x ← arbitrary      ← Generate some arbitrary (unconstrained) x
if (lo < x < hi) then  ← Check if lo < x < hi
    ...
    return some (Node x l r)
else
    return none
```

**Bad!**

**For arbitrary  $x$ ,  $\mathbb{P}(\text{lo} < x < \text{hi})$  is low**

# Generator Schedules

A *smarter* BST generator

do

```
let x ← genSuchThat  
      (fun x ⇒ lo < x < hi)  
let l ← genBST lo x fuel'  
let r ← genBST x hi fuel'  
return (Node x l r))
```

- Prioritize constrained generation (`genSuchThat`) over checks
- Rewrite generator based on variable dependencies (i.e. generate `x` before `l` & `r`)

**Testing Theorems, Fully Automatically**

ANONYMOUS AUTHOR(S)

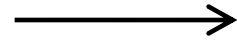
(Elazar-Mittelman et al., submitted to POPL '26)

# Chamelean also derives Checkers

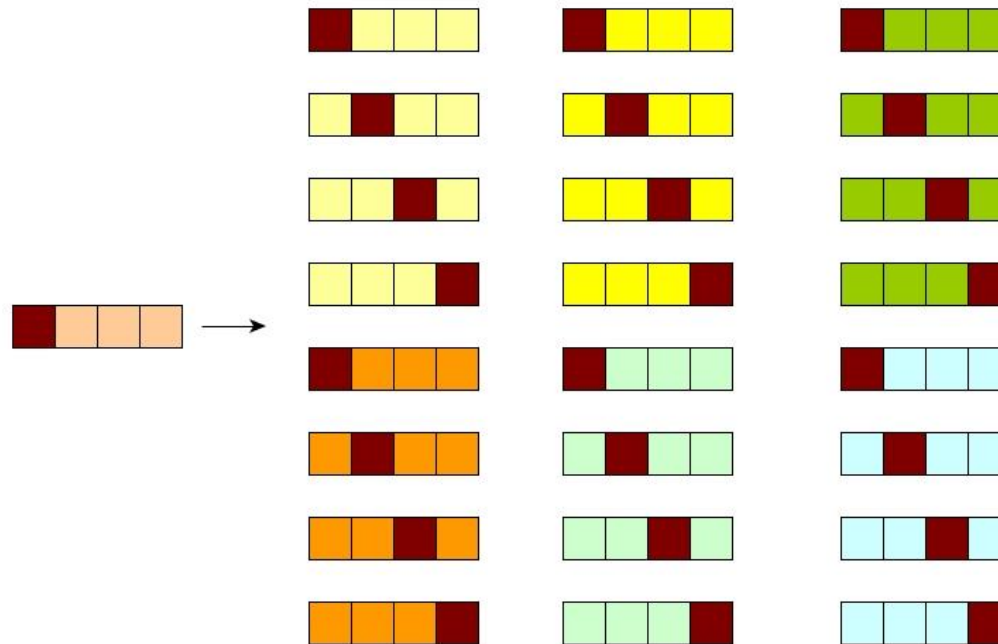
inductive Permutation

(l : List Nat)

(l' : List Nat) : Prop



Chamelean derives a function which *checks*  
if l, l' are permutations of each other



# Chamelean also derives **Enumerators**

**inductive** **Permutation** : List Nat → List Nat → **Prop** where

| **Transitivity** :  $\forall l_1 l_2 l_3,$   
    Permutation  $l_1 l_2 \rightarrow$   
    Permutation  $l_2 l_3 \rightarrow$   
    Permutation  $l_1 l_3$

...

$l_2$  doesn't appear in the conclusion of **Transitivity**

$\Rightarrow$  Chamelean **enumerates**  $l_2$  such that  $l_2$  is also a valid **Permutation**

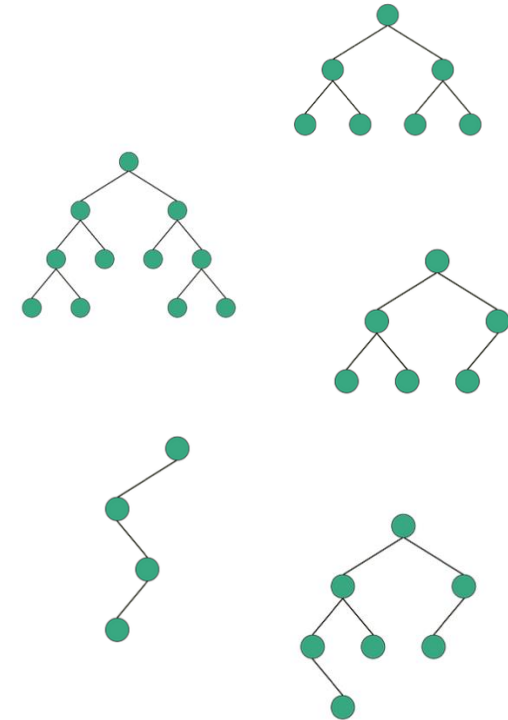
# Deriving Unconstrained Generators for Inductive Types

inductive Tree

| Leaf : Tree

| Node : Nat → Tree → Tree → Tree

deriving Arbitrary, Enum



# Chamelean implements ideas pioneered in Rocq's QuickChick PBT framework

## Testing Theorems, Fully Automatically

ANONYMOUS AUTHOR(S)

Submitted to  
POPL '26



## Computing Correctly with Inductive Relations

Zoe Paraskevopoulou  
z.paraskevopoulou@northeastern.edu  
Northeastern University  
USA

Aaron Eline  
aeline@umd.edu  
University of Maryland, College Park  
USA

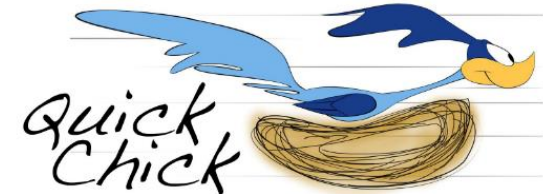
Leonidas Lampropoulos  
leonidas@umd.edu  
University of Maryland, College Park  
USA

PLDI '22

## Generating Good Generators for Inductive Relations

LEONIDAS LAMPROPOULOS, University of Pennsylvania, USA  
ZOE PARASKEVOPOULOU, Princeton University, USA  
BENJAMIN C. PIERCE, University of Pennsylvania, USA

POPL '18



# Demo





```

/-- Datatype for binary trees -/
inductive Tree where
| Leaf : Tree
| Node : Nat → Tree → Tree → Tree

/-- `Between lo x hi` means `lo < x < hi` -/
inductive Between : Nat → Nat → Nat → Prop where
| BetweenN : ∀ n m,
  n ≤ m →
  Between n (.succ n) (.succ (.succ m))
| BetweenS : ∀ n m o,
  Between n m o → Between n (.succ m) (.succ o)

#derive_generator (fun (x : Nat) ⇒ Between lo x hi)

/-- `BST lo hi t` describes whether a tree `t` is a BST that
| contains values strictly within `lo` and `hi` -/
inductive BST : Nat → Nat → Tree → Prop where
| BSTLeaf: ∀ lo hi, BST lo hi .Leaf
| BSTNode: ∀ lo hi x l r,
  Between lo x hi →
  BST lo x l →
  BST x hi r →
  BST lo hi (.Node x l r)

#derive_generator (fun (t : Tree) ⇒ BST lo hi t)

```

Derives a generator for  
**x** satisfying **lo < x < hi**

Derives a generator for  
**Trees** satisfying **BST**

# Code for derived generator automatically displayed in VS Code side panel



The image shows the VS Code interface with two panels. The left panel displays a Lean file named `FinalDemo.lean` with the following code:

```
4  /-- Datatype for binary trees -/
5  inductive Tree where
6  | Leaf : Tree
7  | Node : Nat → Tree → Tree → Tree
8
9  /-- `Between lo x hi` means `lo < x < hi` -/
10 inductive Between : Nat → Nat → Nat → Prop where
11 | BetweenN : ∀ n m,
12   n ≤ m →
13   Between n (.succ n) (.succ (.succ m))
14 | BetweenS : ∀ n m o,
15   Between n m o → Between n (.succ m) (.succ o)
16
17 #derive_generator (fun (x : Nat) => Between lo x hi)
18
19 /-- `BST lo hi t` describes whether a tree `t` is a BST that
20 | contains values strictly within `lo` and `hi` -/
21 inductive BST : Nat → Nat → Tree → Prop where
22 | BSTLeaf: ∀ lo hi, BST lo hi .Leaf
23 | BSTNode: ∀ lo hi x l r,
24   Between lo x hi →
25   BST lo x l →
26   BST x hi r →
27   BST lo hi (.Node x l r)
28
29 #derive_generator (fun (t : Tree) => BST lo hi t)
30
```

The right panel shows the `Lean InfoView` for the file, displaying the generated code for the `#derive_generator` directives:

```
▼ FinalDemo.lean:29:49
▼ Suggestions

Try this generator: instance : ArbitrarySizedSuchThat Tree (fun t_1 => BST lo_1 hi_1 t_1) where
arbitrarySizedST :=
  let rec aux_arb (initSize : Nat) (size : Nat) (lo_1 : Nat) (hi_1 : Nat) : OptionT Plausible.Gen Tree :=
    match size with
    | Nat.zero => OptionTGen.backtrack [(1, return Tree.Leaf)]
    | Nat.succ size' =>
      OptionTGen.backtrack
        [(1, return Tree.Leaf),
         (Nat.succ size', do
           let x <- ArbitrarySizedSuchThat.arbitrarySizedST (fun x => Between lo_1 x hi_1) initSize;
           do
             let l <- aux_arb initSize size' lo_1 x;
             do
               let r <- aux_arb initSize size' x hi_1;
               return Tree.Node x l r)]
  fun size => aux_arb size size lo_1 hi_1

► Messages (1)
► All Messages (2)
```

Clicking on suggested code automatically inserts the generator into your Lean file

```
/-- `BST lo hi t` describes whether a tree `t` is a BST that
| contains values strictly within `lo` and `hi` -/
inductive BST : Nat → Nat → Tree → Prop where
| BSTLeaf: ∀ lo hi, BST lo hi .Leaf
| BSTNode: ∀ lo hi x l r,
  Between lo x hi →
  BST lo x l →
  BST x hi r →
  BST lo hi (.Node x l r)

instance {lo_1 hi_1} : ArbitrarySizedSuchThat Tree (fun t_1 ⇒ BST lo_1 hi_1 t_1) where
  arbitrarySizedST :=
    let rec aux_arb (initSize : Nat) (size : Nat) (lo_1 : Nat) (hi_1 : Nat) : OptionT Plausible.Gen Tree :=
      match size with
      | Nat.zero ⇒ OptionTGen.backtrack [(1, return Tree.Leaf)]
      | Nat.succ size' ⇒
        OptionTGen.backtrack
          [(1, return Tree.Leaf),
           (Nat.succ size', do
             let x ← ArbitrarySizedSuchThat.arbitrarySizedST (fun x ⇒ Between lo_1 x hi_1) initSize;
             do
               let l ← aux_arb initSize size' lo_1 x;
               do
                 let r ← aux_arb initSize size' x hi_1;
                 return Tree.Node x l r)]
    fun size ⇒ aux_arb size size lo_1 hi_1
```

Inserted  
automatically

# Testing a property using Chamelean

Property

$\forall x \text{ lo hi tree,}$   
 $\text{BST lo hi tree} \wedge \text{lo} < x < \text{hi}$   
 $\Rightarrow \text{BST lo hi (insert x tree)}$

Test harness  
(pseudocode)

```
let x ← chooseNat lo hi
let t ← genSuchThat
      (fun tree ⇒ BST lo hi tree)
let t' := insert x t
check (BST lo hi t')
```

} Derived  
generator

← Derived  
checker

(obtained the same  
way as generators)

# Testing a property using Chamelean

```
-- Inserts an element into a tree, respecting the BST invariants -/
def insert (x : Nat) (t : Tree) : Tree :=
  match t with
  | .Leaf ⇒ .Node x .Leaf .Leaf
  | .Node y l r ⇒
    if x < y then
      .Node y (insert x l) r
    else if x > y then
      .Node y l (insert x r)
    else t

-- Test harness for testing the property
`∀ (x : Nat) (t : Tree), BST 0 10 t → BST 0 10 (insert x t)`
for `numTrials` iterations.

-- (Details omitted) -/
def runTests (numTrials : Nat) : IO Unit := ...

-- Uncomment this to run the aforementioned test harness
#eval runTests (numTrials := 10000)
```

## ▼ Messages (1)

### ▼ FinalDemo.lean:82:0

Chamelean: finished 10000 tests, 10000 passed

## ► All Messages (5)

# Falsifying a property using Chamelean

Buggy BST  
insertion function



```
/-- Buggy insertion function: ignores the input tree and
    returns a two-node tree where both values are `x` -/
def buggyInsert (x : Nat) (_ : Tree) : Tree :=
  .Node x (.Node x .Leaf .Leaf) .Leaf

/-- Test harness for testing the property
    `∀ (x : Nat) (t : Tree), BST 0 10 t → BST 0 10 (insert x t)`
    for `numTrials` iterations.

    (Details omitted) -/
def runTests (numTrials : Nat) : IO Unit := ...

-- Uncomment this to run the aforementioned test harness
#eval runTests (numTrials := 10000)
```

▼ Messages (1)

▼ FinalDemo.lean:84:0

Property falsified!

t = Tree.Node 9 (Tree.Node 8 (Tree.Node 7 (Tree.Node 2 (Tree.Leaf)  
(Tree.Leaf)) (Tree.Leaf)) (Tree.Leaf)) (Tree.Leaf)

x = 4

t' = Tree.Node 4 (Tree.Node 4 (Tree.Leaf) (Tree.Leaf)) (Tree.Leaf)

► All Messages (5)



# Case Studies



# Examples

$\Gamma \vdash e : \tau$

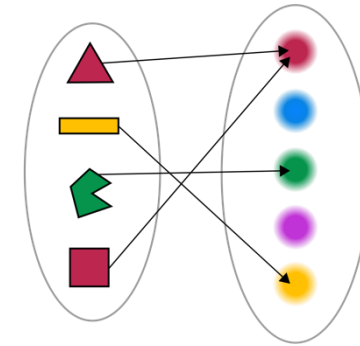
Well-typed  
STLC terms



Binary trees  
(BSTs, balanced, complete, ...)

$[^*?@[^*?\.[^*]$

Strings that match  
regular expressions



API calls to a  
Key-Value Store



# Generating Simply-Typed $\lambda$ -Calculus Terms

This is an area of active research:

Pałka et al. AST '11  
Fetscher et al. ESOP '15  
Claessen et al. JFP '15  
Midtgaard et al. ICFP '17  
Frank et al. POPL '24

...

Chameleon automatically derives a generator for well-typed terms!

```
 $\lambda x : \text{Nat. } 1$   
 $\lambda x : \text{Nat. } ((3 + x) + 4)$   
 $((\lambda x : \text{Nat. } x) 0) + ((\lambda x : \text{Nat. } x + 4) 1)$ 
```

# Aside: the Cedar language

Cutler et al. OOPSLA '24  
Disselkoen et al. FSE '24

DSL developed at AWS  
for rule-based access control:

```
forbid(  
  principal,  
  action,  
  resource)  
  
when {  
  resource.tags.contains("private") &&  
  !(resource in principal.account)  
};
```

// Effect  
// Scope  
  
// Condition

# Ongoing Work: Generating Cedar Terms

$\alpha; \Gamma \vdash \text{true} : \text{True}; \emptyset$	$\alpha; \Gamma \vdash e_1 : E_1; \varepsilon_1$	$s_1 \neq s_2$
$\alpha; \Gamma \vdash \text{false} : \text{False}; \varepsilon$	$\alpha; \Gamma \vdash e_2 : E_2; \varepsilon_2$	$E_1 \neq E_2$
$\alpha; \Gamma \vdash e == e : \text{True}; \emptyset$	$\alpha; \Gamma \vdash e_1 == e_2 : \text{False}; \varepsilon$	$\alpha; \Gamma \vdash E::s_1 == E::s_2 : \text{False}; \varepsilon$
$\alpha; \Gamma \vdash e_1 : \tau_1; \varepsilon_1$	$\tau_1 <: \tau$	$\alpha; \Gamma \vdash e : E_1; \varepsilon$
$\alpha; \Gamma \vdash e_2 : \tau_2; \varepsilon_2$	for some $\tau$	$(E_1 = E) \Rightarrow \tau = \text{True}$
$\alpha; \Gamma \vdash e_1 == e_2 : \text{Bool}; \emptyset$		$(E_1 \neq E) \Rightarrow \tau = \text{False}$
$\alpha; \Gamma \vdash e_1 : E_1; \varepsilon_1$	$\alpha; \Gamma \vdash e_2 : E_2; \varepsilon_2$	
$E_1 \neq E_2$	$M(E_1) = (\_, H)$	$E_2 \notin H$
$\alpha; \Gamma \vdash e_1 \text{ in } e_2 : \text{False}; \varepsilon$	$\alpha; \Gamma \vdash e_1 : E_1; \varepsilon_1$	$\alpha; \Gamma \vdash e_2 : E_2; \varepsilon_2$
$\alpha; \Gamma \vdash e_1 : \text{True}; \varepsilon_1$	$\alpha \cup \varepsilon_1; \Gamma \vdash e_2 : \tau; \varepsilon_2$	$\alpha; \Gamma \vdash e_1 : \text{False}; \varepsilon_1$
$\alpha; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau; \varepsilon_1 \cup \varepsilon_2$		$\alpha; \Gamma \vdash e_3 : \tau; \varepsilon_3$
$\alpha; \Gamma \vdash e_1 : \text{Bool}; \varepsilon_1$	$\alpha \cup \varepsilon_1; \Gamma \vdash e_2 : \tau; \varepsilon_2$	$\alpha; \Gamma \vdash e_3 : \tau; \varepsilon_3$
$\alpha; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau, (\varepsilon_1 \cup \varepsilon_2) \cap \varepsilon_3$		$\alpha; \Gamma \vdash e : \tau; \varepsilon$
$\alpha; \Gamma \vdash e : \tau; \varepsilon$	$\alpha; \Gamma \vdash e : \tau; \varepsilon$	$\text{attribute}(f, \tau) = \omega f : \tau_f \text{ when}$
$\text{attribute}(f, \tau) = (? f : \tau_f)$	$\text{attribute}(f, \tau) = (\omega f : \tau_f)$	$\tau = \{..., \omega f : \tau_f, ...\}, \text{ or}$
$\alpha; \Gamma \vdash e \text{ has } f : \text{Bool}, \{e.f\}$	$\omega = ? \Rightarrow e.f \in \alpha$	$\tau = E \text{ and}$
$\alpha; \Gamma \vdash e.f : \tau_f; \emptyset$		$M(E) = (\{..., \omega f : \tau_f, ...\}, \_)$

Lean formalization of Cedar's *static* semantics:

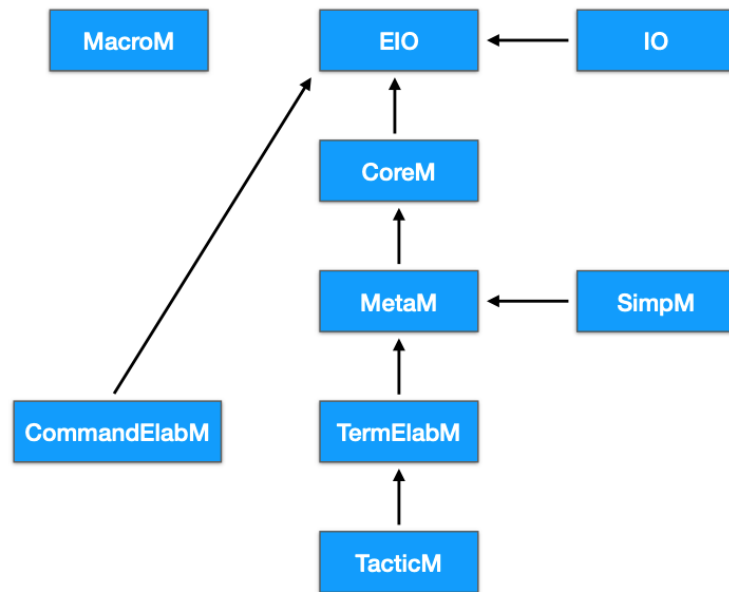
- 29 inductive relations
- Syntax defined via 17 types
- Chamelean can handle 23 / 41 typing rules
- Struggles with typing rules involving complex constraints
  - (Algorithm times out)

Cutler et al. OOPSLA '24

# Building Chamelean via Metaprogramming

- Before internship: Chamelean prototype produced Lean code via pretty-printed strings
- My job: Implement QuickChick's algorithms using Lean metaprogramming idioms

## Lean Monad Zoo



My prior research: developing PBT tools using OCaml metaprogramming

### **MICA: Automated Differential Testing for OCaml Modules**

ERNEST NG\*, University of Pennsylvania and Cornell University, USA

HARRISON GOLDSTEIN\*, University of Pennsylvania and University of Maryland, USA

BENJAMIN C. PIERCE, University of Pennsylvania, USA

OCaml Workshop '24

# Future / Related Work

# Future Work

Automatically produce correctness proofs for derived generators

Soundness: If  is generated, then  is a valid BST

Completeness: All BSTs can be generated 

# Future Work

Integrate Chamelean with PBT projects by collaborators at UMD



Derive generators using Lean's Aesop proof search tactic

Heuristics for optimizing derived generators

**The Search for Constrained Random Generators**

ANONYMOUS AUTHOR(S)

**Testing Theorems, Fully Automatically**

ANONYMOUS AUTHOR(S)

Both submitted to POPL '26

# Future Work

Give users greater control over the distribution of generated values

Allow users to assign probabilities to constructors (à la OCaml QuickCheck)

```
type tree =  
  | Leaf  
  | Node1 of tree * int * tree [@weight 1/2]  
  | Node2 of tree * int * tree [@weight 1/3]  
[@@deriving quickcheck]
```

Tune generators using ideas from probabilistic programming

## **[DRAFT] Tuning Random Generators**

Property-Based Testing as Probabilistic Programming

RYAN TJOA, University of Washington, USA

POORVA GARG, University of California, Los Angeles, USA

HARRISON GOLDSTEIN, University of Maryland, USA

TODD MILLSTEIN, University of California, Los Angeles, USA

BENJAMIN PIERCE, University of Pennsylvania, USA

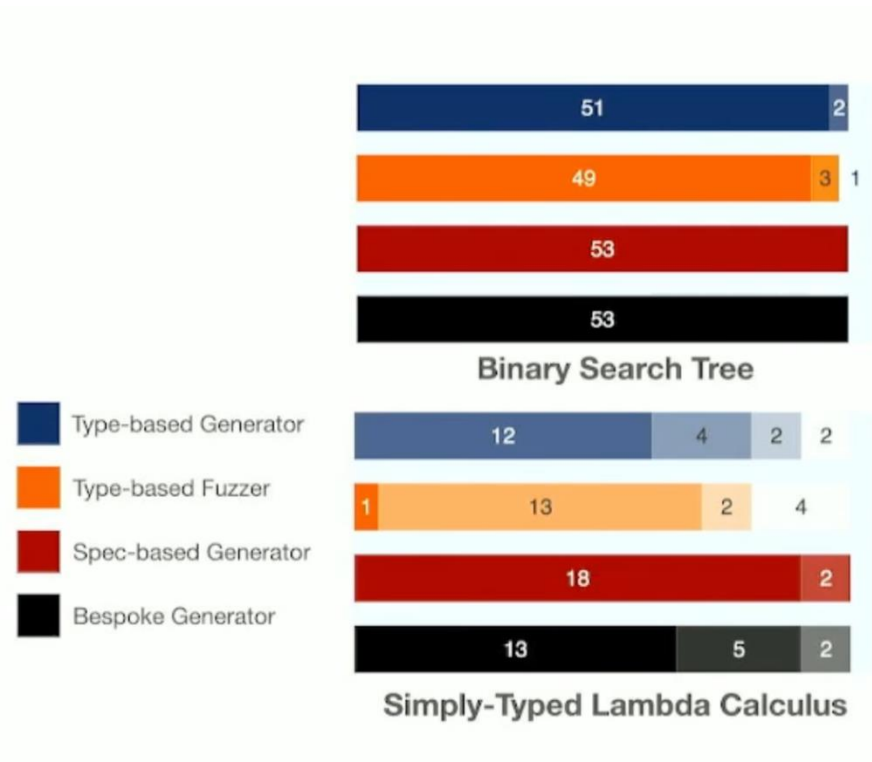
GUY VAN DEN BROECK, University of California, Los Angeles, USA

To appear at OOPSLA '25



# Future Work

Benchmark Chamelean's derived generators against their QuickChick counterparts



## ETNA: An Evaluation Platform for Property-Based Testing (Experience Report)

JESSICA SHI, University of Pennsylvania, USA

ALPEREN KELES, University of Maryland, USA

HARRISON GOLDSTEIN, University of Pennsylvania, USA

BENJAMIN C. PIERCE, University of Pennsylvania, USA

LEONIDAS LAMPROPOULOS, University of Maryland, USA

ICFP '23

(extended version submitted to JFP)

# Summary

Chamelean derives the following  
using Lean metaprogramming idioms:

**For inductive relations**

Generators

Enumerators

Checkers

**For inductive types**

Generators

Enumerators

# Chamelean is open-source!

ngernest/chamelean



Ongoing work:  
merging Chamelean into Lean's Plausible PBT library

**Feat: Automatically Derive Generators for Algebraic Data Types #35**

 **Open** ngernest wants to merge 60 commits into `leanprover-community:main` from `ngernest:main` 

# Thank you!

Ernest Ng

[ernest@cs.cornell.edu](mailto:ernest@cs.cornell.edu)



# Appendix



# Checkers Invoking Enumerators: An Example

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

```
-- `typing Γ e τ` is the typing judgment `Γ ⊢ e : τ`  
inductive typing: context → term → type → Prop where  
  | TApp : ∀ Γ e1 e2 τ1 τ2,  
    typing Γ e2 τ1 →  
    typing Γ e1 (Fun τ1 τ2) →  
    typing Γ (App e1 e2) τ2  
  ...
```

**τ1** doesn't appear in the conclusion of **TApp**, but is used in hypotheses  
⇒ the derived checker for **typing** needs to enumerate **τ1**

# Checkers Invoking Enumerators: An Example

```
| TApp :  $\forall \Gamma \ e1 \ e2 \ \tau1 \ \tau2,$   
    typing  $\Gamma \ e2 \ \tau1 \rightarrow$   
    typing  $\Gamma \ e1 \ (\text{Fun } \tau1 \ \tau2) \rightarrow$   
    typing  $\Gamma \ (\text{App } e1 \ e2) \ \tau2$ 
```

```
match e,  $\tau$  with  
| App e1 e2,  $\tau2 \Rightarrow$  do  
    let  $\tau1 \leftarrow$  enumerateSuchThat  
        (fun  $\tau \Rightarrow$  typing  $\Gamma \ e2 \ \tau$ )  
    check (typing  $\Gamma \ e1 \ (\text{Fun } \tau1 \ \tau2)$ )  
| ...
```

$\tau1$  doesn't appear in the conclusion of TApp, but is used in hypotheses  
 $\Rightarrow$  the derived checker for typing needs to enumerate  $\tau1$

Given some inductive relation  $P$ , Chamelean derives:

## Generators

(random)

`Gen (Option  $\alpha$ )`

Produce  $\alpha$ 's  
satisfying  $P$

## Enumerators

(deterministic)

`List (Option  $\alpha$ )`

## Checkers

(semi-decision procedures)

`Option Bool`

(None = out of fuel)

Checks if  
 $P(\alpha)$  holds



# Smarter constraint ordering

```
inductive BST : Nat → Nat → Tree → Prop where
| BSTNode: ∀ lo hi x l r,
    lo < x < hi →
    BST lo x l →
    BST x hi r →
    BST lo hi (Node x l r)
```

Behavior of derived generator :

1. Generate  $x$  such that  $lo < x < hi$
2. Generate left & right subtrees

# Smarter constraint ordering

inductive BST : Nat → Nat → Tree → Prop where

| BSTNode:  $\forall$  lo hi x l r,

BST lo x l →

BST x hi r →

**lo < x < hi →**

-- Order of hypotheses swapped

BST lo hi (Node x l r)

1. Generate some unconstrained x
2. Generate left & right subtrees
3. Check that lo < x < hi

This check will often fail!

(For arbitrarily generated x,  $\text{Pr}[\text{lo} < x < \text{hi}]$  is low)