Chamelean

Property-Based Testing for Lean via Metaprogramming

Ernest Ng
(advised by Cody Roux & Mike Hicks)

In collaboration with
University of Maryland 1856

Property-Based Testing

1. Write *properties*

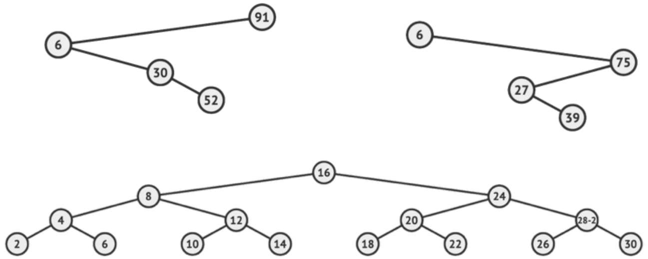
Spec for Binary Search Trees (BSTs):

```
∀ x tree,
  isBST tree
  ⟹ isBST (insert x tree)
```

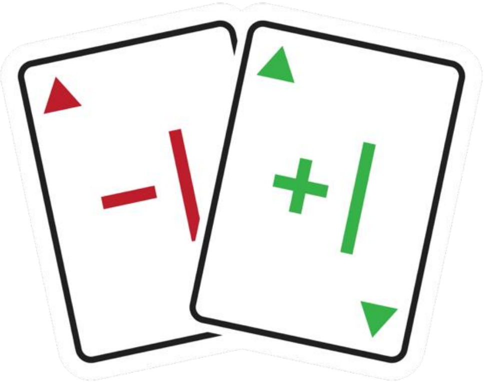
Property-Based Testing (PBT)

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

$\Longrightarrow$ Use PBT to *test* your specs before starting a proof!

> the ability to use the same specifications both for runtime testing and for verification offers potential benefits for both, e.g. for quickly discovering some code and specification errors before embarking on proof,

Banerjee et al. (POPL '25)

Every major proof assistant has a PBT framework

ROCQ

Agda

Isabelle

LEAN

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
  …
```

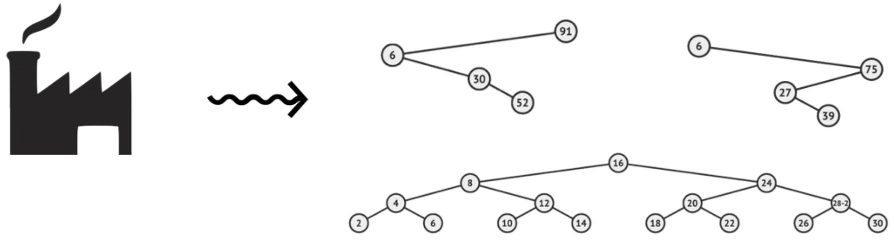
✓ Facilitates inductive reasoning!
✓ Easy to model properties!
✕ No computational content
(Chamelean addresses this issue!)

Chamelean

1. User specifies a Lean inductive relation

```
inductive isBST :
  Nat → Nat → Tree → Prop where
| BSTLeaf : …
| BSTNode : …
```

2. Chamelean derives a generator of random data satisfying the relation



(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 drecreasing)

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
(talk to me offline about the unification algorithm!)

```
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 : ∀ 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)
        …
```

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’
         …
```

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

Recursively generate right subtree 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 : ∀ lo hi x l r,
    lo < x < hi  →
    isBST lo x l →
    isBST x hi r →
    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 ***naive*** BST generator

```
let x ← arbitrary
if (lo < x < hi) then
    …
    return some (Node x l r)
else
    return none
```

Generate some `arbitrary` (unconstrained) x

Check if `lo < x < hi`

Bad!
For arbitrary x, $\mathbb{P}$(`lo < x < 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)

(submitted to POPL '26)

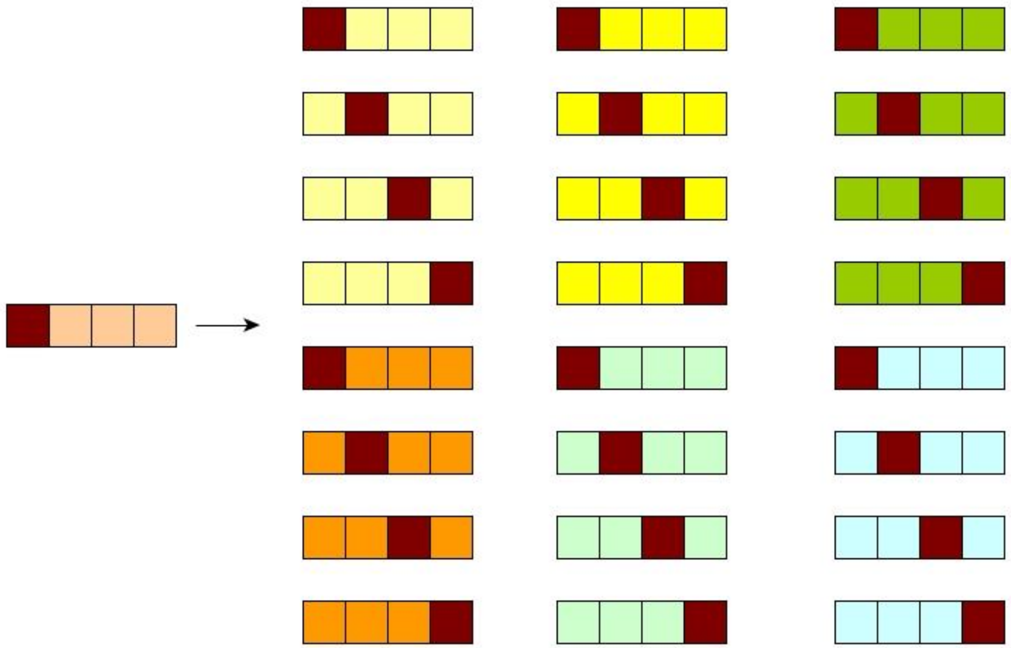
Chamelean also derives **Checkers**

```
inductive Permutation
  (l  : List Nat)
  (l' : List Nat) : Prop
```

Chamelean derives a function which ***checks***

if `l`, `l'` are permutations of each other

(this is a semi-decision procedure)



Chamelean also derives **Enumerators**

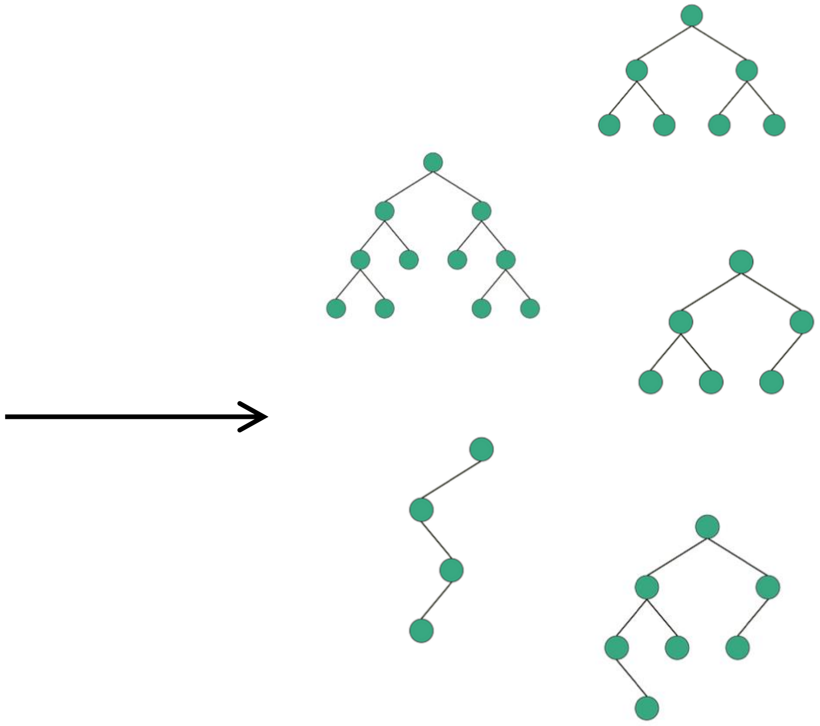
```
inductive Permutation : List Nat → List Nat → Prop where
  | Transitivity : ∀ l₁ l₂ l₃,
              Permutation l₁ l₂ →
              Permutation l₂ l₃ →
              Permutation l₁ l₃
...
```

`l₂` doesn't appear in the conclusion of `Transitivity`

⟹ Chamelean ***enumerates*** `l₂` such that `l₂` 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

(talk to me offline about these papers!)

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 **`Tree`**s satisfying **`BST`**

Code for derived generator automatically displayed in VS Code side panel

FinalDemo.lean U ×

Plausible > Chamelean > FinalDemo.lean > ...

```lean
/-- 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)
```

Lean InfoView ×

▼ FinalDemo.lean:29:49

▼ Suggestions

```lean
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

```
∀ x lo hi tree,
  BST lo hi tree ∧ lo < x < hi
  ⟹ 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

```lean
/-- 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

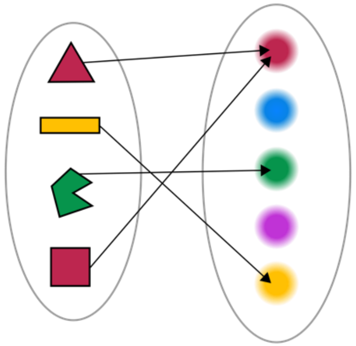
$$\Gamma \vdash e : \tau$$

Well-typed
STLC terms

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

Strings that match
regular expressions

Binary trees
(BSTs, balanced, complete, …)



API calls to a
Key-Value Store

Generating Well-Typed STLC 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
...

Chamelean automatically derives a generator for well-typed terms!

```
λ x : Nat. 1
λ x : Nat. ((3 + x) + 4)
((λ x : Nat. x) 0) + ((λ x : 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:

```
// Example Cedar policy:
// Interns can't create lists
forbid(
     principal in Team::interns,
     action == CreateList,
     resource
);
```

Ongoing Work: Generating Cedar Terms

$$\alpha;\Gamma \vdash \texttt{true} : \mathit{True}; \emptyset \qquad \alpha;\Gamma \vdash \texttt{false} : \mathit{False}; \varepsilon \qquad \alpha;\Gamma \vdash e == e : \mathit{True}; \emptyset$$

$$\frac{\begin{array}{c}\alpha;\Gamma \vdash e_1 : E_1; \varepsilon_1 \\ \alpha;\Gamma \vdash e_2 : E_2; \varepsilon_2 \qquad E_1 \neq E_2\end{array}}{\alpha;\Gamma \vdash e_1 == e_2 : \mathit{False}; \varepsilon} \qquad \frac{s_1 \neq s_2}{\alpha;\Gamma \vdash E{::}s_1 == E{::}s_2 : \mathit{False}; \varepsilon}$$

$$\frac{\begin{array}{cc}\alpha;\Gamma \vdash e_1 : \tau_1; \varepsilon_1 & \tau_1 <: \tau \quad \tau_2 <: \tau \\ \alpha;\Gamma \vdash e_2 : \tau_2; \varepsilon_2 & \text{for some } \tau\end{array}}{\alpha;\Gamma \vdash e_1 == e_2 : \mathit{Bool}; \emptyset} \qquad \frac{\begin{array}{c}\alpha;\Gamma \vdash e : E_1; \varepsilon \\ (E_1 = E) \Rightarrow \tau = \mathit{True} \qquad (E_1 \neq E) \Rightarrow \tau = \mathit{False}\end{array}}{\alpha;\Gamma \vdash e \;\texttt{is}\; E : \tau; \emptyset}$$

$$\frac{\begin{array}{c}\alpha;\Gamma \vdash e_1 : E_1; \varepsilon_1 \qquad \alpha;\Gamma \vdash e_2 : E_2; \varepsilon_2 \\ E_1 \neq E_2 \qquad M(E_1) = (\_, H) \qquad E_2 \notin H\end{array}}{\alpha;\Gamma \vdash e_1 \;\texttt{in}\; e_2 : \mathit{False}; \varepsilon} \qquad \frac{\alpha;\Gamma \vdash e_1 : E_1; \varepsilon_1 \qquad \alpha;\Gamma \vdash e_2 : E_2; \varepsilon_2}{\alpha;\Gamma \vdash e_1 \;\texttt{in}\; e_2 : \mathit{Bool}; \emptyset}$$

$$\frac{\alpha;\Gamma \vdash e_1 : \mathit{True}; \varepsilon_1 \qquad \alpha \cup \varepsilon_1;\Gamma \vdash e_2 : \tau; \varepsilon_2}{\alpha;\Gamma \vdash \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 : \tau; \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\alpha;\Gamma \vdash e_1 : \mathit{False}; \varepsilon_1 \qquad \alpha;\Gamma \vdash e_3 : \tau; \varepsilon_3}{\alpha;\Gamma \vdash \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 : \tau; \varepsilon_3}$$

$$\frac{\alpha;\Gamma \vdash e_1 : \mathit{Bool}; \varepsilon_1 \qquad \alpha \cup \varepsilon_1;\Gamma \vdash e_2 : \tau; \varepsilon_2 \qquad \alpha;\Gamma \vdash e_3 : \tau; \varepsilon_3}{\alpha;\Gamma \vdash \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 : \tau, (\varepsilon_1 \cup \varepsilon_2) \cap \varepsilon_3} \qquad \frac{\alpha;\Gamma \vdash e : \tau; \varepsilon \qquad \tau <: \tau'}{\alpha;\Gamma \vdash e : \tau'; \varepsilon}$$

$$\frac{\begin{array}{c}\alpha;\Gamma \vdash e : \tau; \varepsilon \\ \mathit{attribute}(f, \tau) = \;(?\, f : \tau_f)\end{array}}{\alpha;\Gamma \vdash e \;\mathit{has}\; f : \mathit{Bool}, \{e.f\}} \qquad \frac{\begin{array}{c}\alpha;\Gamma \vdash e : \tau; \varepsilon \\ \mathit{attribute}(f, \tau) = (\omega f : \tau_f) \\ \omega = ? \Rightarrow e.f \in \alpha\end{array}}{\alpha;\Gamma \vdash e.f : \tau_f; \emptyset}$$

$$\begin{array}{l}\mathit{attribute}(f, \tau) = \omega f : \tau_f \text{ when} \\ \quad \tau = \{\ldots, \omega f : \tau_f, \ldots\}, \text{ or} \\ \quad \tau = E \text{ and} \\ \qquad M(E) = (\{\ldots, \omega f : \tau_f, \ldots\}, \_)\end{array}$$

Cutler et al. OOPSLA '24

Lean formalization of Cedar's *static* semantics:

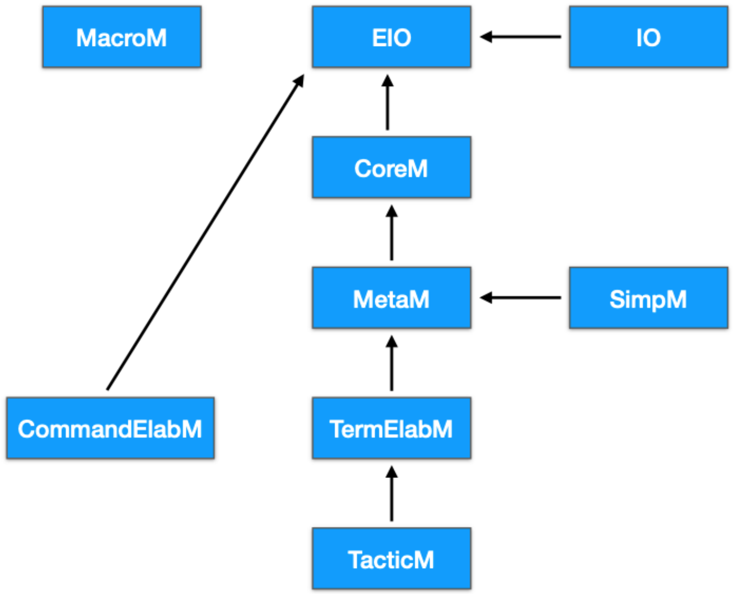
- 29 inductive relations
- Syntax defined via 17 types

- Chamelean can handle 23 / 41 typing rules
- Struggles with typing rules involving complex constraints
    - Algorithm times out! (Talk to me offline for details)

Building Chamelean via Metaprogamming

- 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

The Golden Age of PBT Research

6 PBT papers to appear at ICFP / OOPSLA '25!

Bennet: Randomized Specification Testing for Heap-Manipulating Programs

ZAIN K AAMER, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

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 C. PIERCE, University of Pennsylvania, USA
GUY VAN DEN BROECK, University of California, Los Angeles, USA

We've Got You Covered: Type-Guided Repair of Incomplete Input Generators

PATRICK LAFONTAINE, Purdue University, USA
ZHE ZHOU, Purdue University, USA
ASHISH MISHRA, IIT Hyderabad, India
SURESH JAGANNATHAN, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA

Teaching Software Specification (Experience Report)

CAMERON MOY, Northeastern University, USA
DANIEL PATTERSON, Northeastern University, USA

Lightweight Testing of Persistent Amortized Time Complexity in the Credit Monad

Technical Report, Aug 19, 2025 (v4).

Anton Lorenzen
University of Edinburgh
Edinburgh, UK
anton.lorenzen@ed.ac.uk

An Empirical Evaluation of Property-Based Testing in Python

SAVITHA RAVI, University of California San Diego, USA
MICHAEL COBLENZ, University of California San Diego, USA

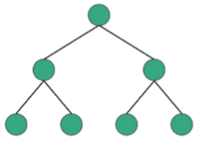
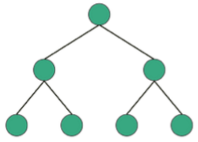
I maintain a PBT bibliography on GitHub:

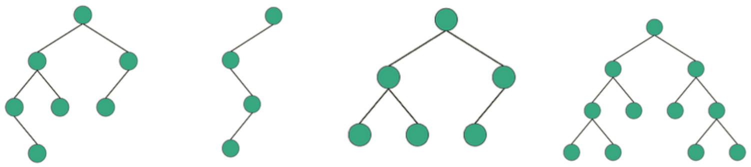
`ngernest/pbt-bibliography`

Future Work

Automatically produce correctness proofs for derived generators

(ask me offline about how we prove this!)

Soundness: If  is generated, then  is a valid BST

Completeness: All BSTs can be generated 

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 the Dice probabilistic language

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 C. PIERCE, University of Pennsylvania, USA
GUY VAN DEN BROECK, University of California, Los Angeles, USA

To appear at OOPSLA '25

Future Work

Integrate Chamelean with PBT
projects by our collaborators

Derive generators using Lean's
Aesop proof search tactic

The Search for Constrained Random Generators

ANONYMOUS AUTHOR(S)

Heuristics for optimizing
derived generators

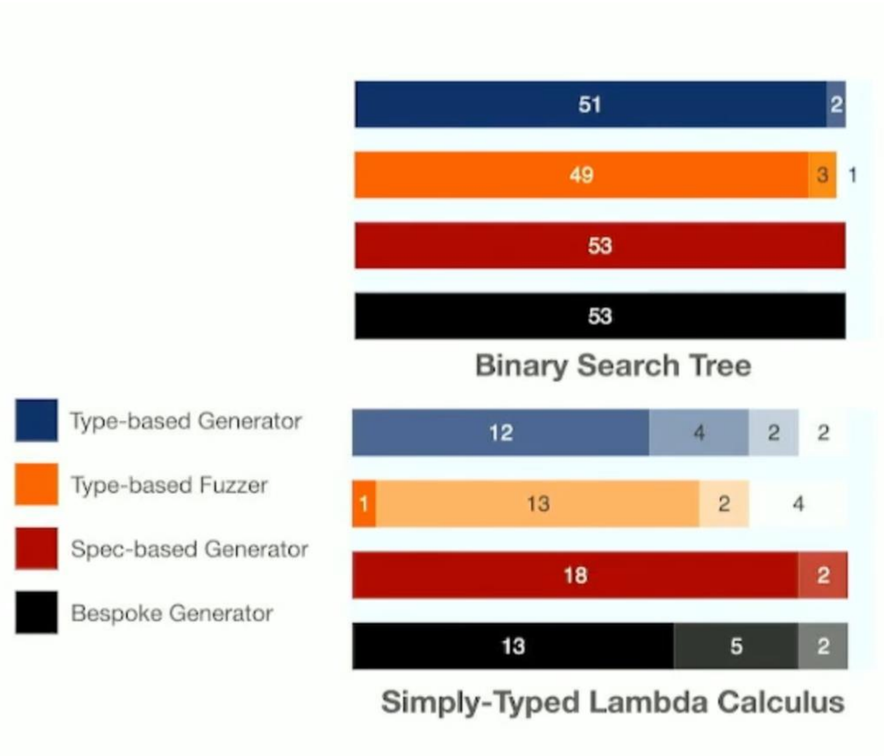
Testing Theorems, Fully Automatically

ANONYMOUS AUTHOR(S)

Both submitted to POPL '26

Future Work

Benchmark Chamelean's derived generators against their QuickChick counterparts



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

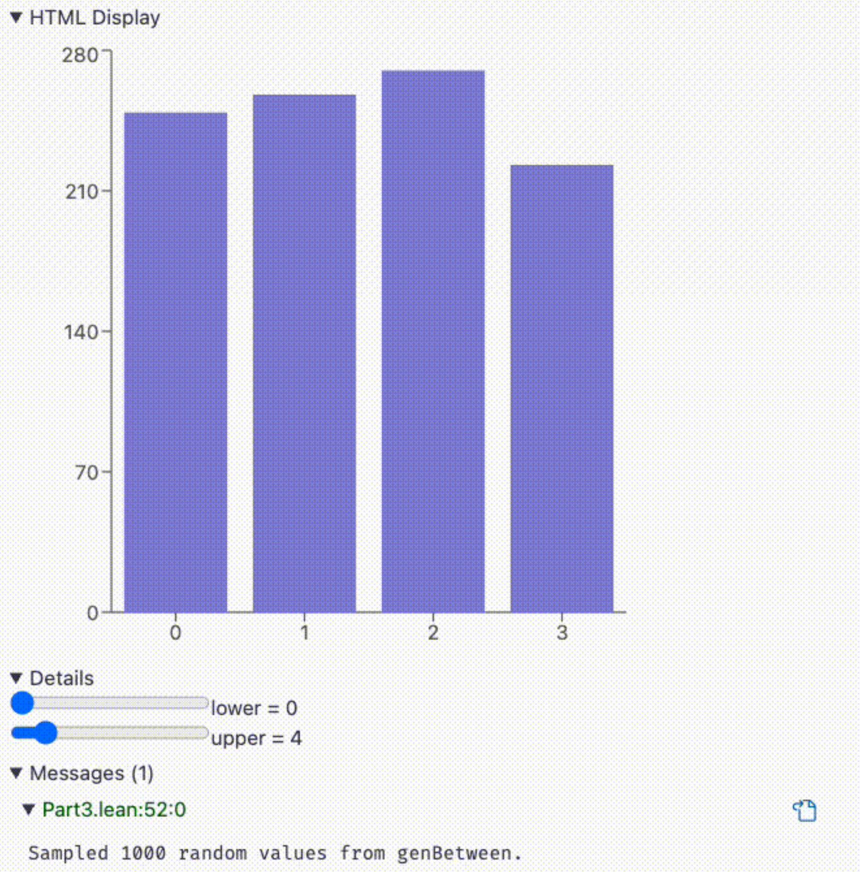
JESSICA SHI, University of Pennsylvania, USA
ALPEREN KELES, University of Maryland, USA
HARRISON GOLDSTEIN, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

ICFP '23
(extended version submitted to JFP)

Use Lean's support for live programming to give users greater insight into generated values

```
-- Generates a `Nat` uniformly at random between `lower` and `upper`
-- Dragging the slider in VS Code changes the distribution
-- (note how the code also changes!)
def genBetween : Gen Nat := do
  Gen.chooseNatLt [slider|lower=0] [slider|upper=4] (by simp)

#view_distribution genBetween
```



Example from Harry Goldstein

Future Work

Extend Tyche with support for Chamelean
(VS Code extension for visualizing PBT effectiveness)

Tyche: Making Sense of Property-Based Testing Effectiveness

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Jeffrey Tao
University of Pennsylvania
Philadelphia, PA, USA
jefftao@seas.upenn.edu

Zac Hatfield-Dodds*
Anthropic
San Francisco, CA, USA
zac.hatfield.dodds@gmail.com

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

Goldstein et al. UIST '24

Summary

Chamelean derives the following
using Lean metaprogramming idioms:

For inductive relations

Generators

Enumerators

Checkers

For algebraic data 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

Appendix

Inductive Relations Can Encode Non-Termination

```
-- A contrived example: `isZero` holds for `0` and no other `Nat`
inductive isZero : Nat → Prop where
  | Zero : isZero 0
  | NonZero : ∀ n, isZero (succ n) → isZero n
```

Example from Paraskevopoulou et al. (PLDI '22), §5

- The derived checker always returns `None` for any non-zero `Nat` (regardless of how much fuel is supplied)
- It will try to satisfy the `isZero (succ n)` hypothesis of the `NonZero` constructor (which is never satisfied!) until it runs out of fuel

Inductive Relations Can Encode Non-Determinism

Small-step semantics for the untyped λ-calculus, extended with non-deterministic choice `e1 □ e2`

$$\frac{}{e_1 \Box e_2 \longrightarrow e_1}\ (\textsc{SChooseLeft}) \qquad \frac{}{e_1 \Box e_2 \longrightarrow e_2}\ (\textsc{SChooseRight})$$

Example from CS 6110 Lecture 5

Functions vs Inductive Relations (Cont.)

Functions	**(Inductive) Relations**
f : A -> B	**r : A -> B -> Prop**
Total: for every **x : A**, there is one **f x : B** (termination checker)	**Partial:** for every **x : A**, there may be zero or more **y : B** such that **r x y**
In Coq: functions are **computable** ~~**terminates : TuringMachine -> Input -> bool**~~ ~~**equal : ℝ -> ℝ -> bool**~~	May be **non-computable** **terminates : TuringMachine -> Input -> Prop** **equal : ℝ -> ℝ -> Prop**
Proving f x = y is **automatic** if x,y are constants (simpl, reflexivity)	Proving r x y is **manual** even if x,y are constants (apply constructors of r)
eval : env -> stmt -> env	**big_step : env -> stmt -> env -> Prop**

Taken from CS 6115 (Fall '24) Lecture 7 slides

Chamelean / QuickChick's Unification Algorithm

$$
\begin{aligned}
\mathit{unify}\ u_1\ u_2 &= \begin{cases} \mathit{return}\ () & \text{if } u_1 = u_2 \\ r_1 \leftarrow \kappa[u_1];\ r_2 \leftarrow \kappa[u_2];\ \mathit{unifyR}\ (u_1, r_1)\ (u_2, r_2) & \text{otherwise} \end{cases} \\
\mathit{unify}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) &= \mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unify}\ u_1\ (C_2\ r_{21}\ \cdots\ r_{2m}) &= r_1 \leftarrow \kappa[u_1];\ \mathit{unifyRC}\ (u_1, r_1)\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unify}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ u_2 &= r_2 \leftarrow \kappa[u_2];\ \mathit{unifyRC}\ (u_2, r_2)\ (C_1\ r_{11}\ \cdots\ r_{1n})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{unifyR}\ (u_1, \mathit{undef}_\tau)\ (u_2, r) &= \mathit{update}\ u_1\ u_2 \\
\mathit{unifyR}\ (u_1, r)\ (u_2, \mathit{undef}_\tau) &= \mathit{update}\ u_2\ u_1 \\
\mathit{unifyR}\ (u_1, u_1')\ (u_2, r) &= \mathit{unify}\ u_1'\ u_2 \\
\mathit{unifyR}\ (u_1, r)\ (u_2, u_2') &= \mathit{unify}\ u_1\ u_2' \\
\mathit{unifyR}\ (\_, C_1\ r_{11}\ \cdots\ r_{1n})\ (\_, C_2\ r_{21}\ \cdots\ r_{2m}) &= \mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyR}\ (u_1, \mathit{fixed})\ (u_2, \mathit{fixed}) &= \mathit{equality}\ u_1\ u_2;\ \mathit{update}\ u_1\ u_2 \\
\mathit{unifyR}\ (u_1, \mathit{fixed})\ (\_, C_2\ r_{21}\ \cdots\ r_{2m}) &= \mathit{match}\ u_1\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyR}\ (\_, C_1\ r_{11}\ \cdots\ r_{1n})\ (u_2, \mathit{fixed}) &= \mathit{match}\ u_2\ (C_1\ r_{11}\ \cdots\ r_{1n})
\end{aligned}
$$

$$
\mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) = \begin{cases} \mathit{fold}\ \mathit{unify}\ \overline{(r_{1i}, r_{2i})} & \text{if } C_1 = C_2 \text{ and } n = m \\ \bot & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathit{unifyRC}\ (u, \mathit{undef}_\tau)\ (C_2\ r_{21}\ \cdots\ r_{2m}) &= \mathit{update}\ u_1\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyRC}\ (u, u')\ (C_2\ r_{21}\ \cdots\ r_{2m}) &= r \leftarrow \kappa[u'];\ \mathit{unifyRC}\ (u', r)\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyRC}\ (u, \mathit{fixed})\ (C_2\ r_{21}\ \cdots\ r_{2m}) &= \mathit{match}\ u\ (C_2\ r_{21}\ \cdots\ r_{2m}) \\
\mathit{unifyRC}\ (u, C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m}) &= \mathit{unifyC}\ (C_1\ r_{11}\ \cdots\ r_{1n})\ (C_2\ r_{21}\ \cdots\ r_{2m})
\end{aligned}
$$

$$
\mathit{match}\ u\ (C\ r_1\ \cdots\ r_n) = \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{pattern}\ u\ (C\ \overline{p})
$$

$$
\begin{aligned}
\mathit{matchAux}\ (C\ \overline{r}) &= \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{return}\ (C\ \overline{p}) \\
\mathit{matchAux}\ u &= r \leftarrow \kappa[u];\ \mathit{case}\ r\ \mathit{of}\ \ \mathit{undef}_\tau \Rightarrow \mathit{update}\ u\ \mathit{fixed} \\
&\qquad \mid \mathit{fixed} \Rightarrow u' \leftarrow \mathit{fresh};\ \mathit{equality}\ u'\ u;\ \mathit{update}\ u'\ u;\ \mathit{return}\ u' \\
&\qquad \mid u' \Rightarrow \mathit{matchAux}\ u' \\
&\qquad \mid C\ \overline{r} \Rightarrow \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{return}\ (C\ \overline{p})
\end{aligned}
$$

Fig. 3. Unification algorithm

Lampropoulos et al. POPL '18

- Works on ***unknowns*** (set of values that variables can take on during generation)
  - Difference: unknowns can be provided as *inputs* to generators (i.e. they have one single fixed value, but that value is unknown at compile time)

- Implemented using the State & Option monads!

<u>FAQ</u>

Q: Why can't we just use Rocq/Lean's internal unifier?

A: Rocq's unifier works on Rocq terms, whereas we need to unify our representation of knowledge of the state of each unknown at any particular point

General Structure of a Generator

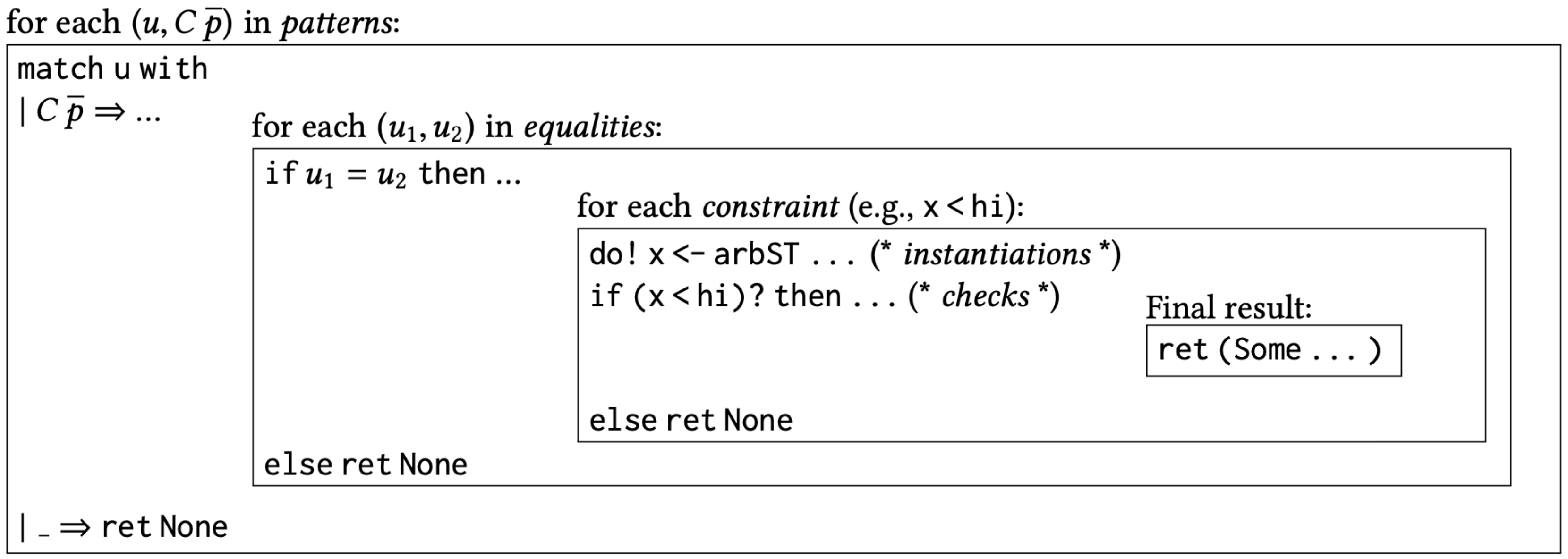


Fig. 1. General Structure of each sub-generator

Lampropoulos et al. POPL '18

Checker Correctness Proof (Sketch)

Soundness

$\forall$ fuel,
  check (P e_1 … e_m) fuel = Some true
  $\Longrightarrow$ P e_1 … e_m

By induction on the `fuel` argument

Completeness

P e_1 … e_m
  $\Longrightarrow$ $\exists$ fuel,
  check (P e_1 … e_m) fuel = Some true

By induction on the derivation of P e_1 … e_m

Proof for generators is more complex (requires reasoning about generators' ***support***)

See §5 of Paraskevopoulou et al. (PLDI '22) for more details!

Checkers Invoking Enumerators: An Example

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \text{TApp}$$

```
-- `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 : ∀ Γ e1 e2 τ1 τ2,
    typing Γ e2 τ1 →
    typing Γ e1 (Fun τ1 τ2) →
    typing Γ (App e1 e2) τ2
```

```
match e, τ with
| App e1 e2, τ2 ⇒ do
  let τ1 ← enumerateSuchThat
             (fun τ ⇒ typing Γ e2 τ)
  check (typing Γ e1 (Fun τ1 τ2))
| …
```

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

Given some inductive relation P, Chamelean derives:

Generators (random)	`Gen (Option α)`	Produce α's satisfying P
Enumerators (deterministic)	`List (Option α)` (produced lazily)	
Checkers (semi-decision procedures)	`Option Bool` (None = out of fuel)	Checks if `P(α)` 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: ∀ 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`, `Pr[lo < x < hi]` is low)