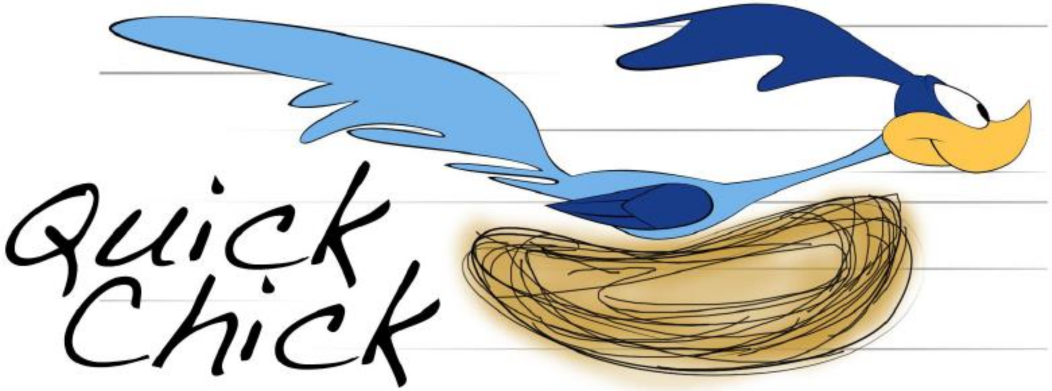
Random Testing in the Coq Proof Assistant

Computational Logic and Applications

Leonidas Lampropoulos

with Zoe Paraskevopoulou and Benjamin C. Pierce

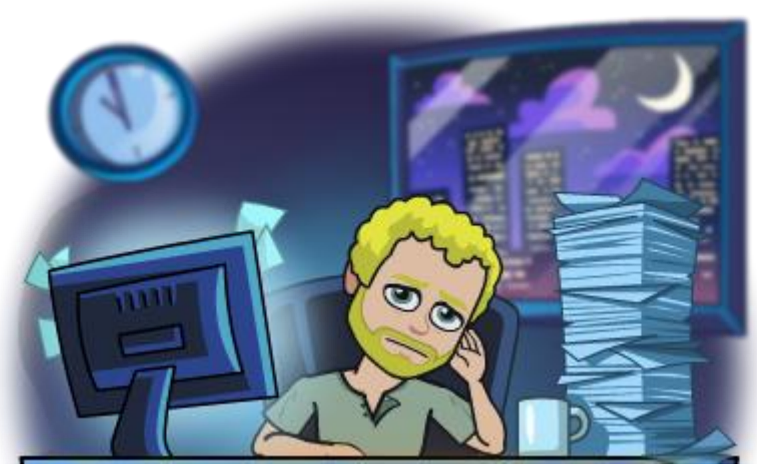


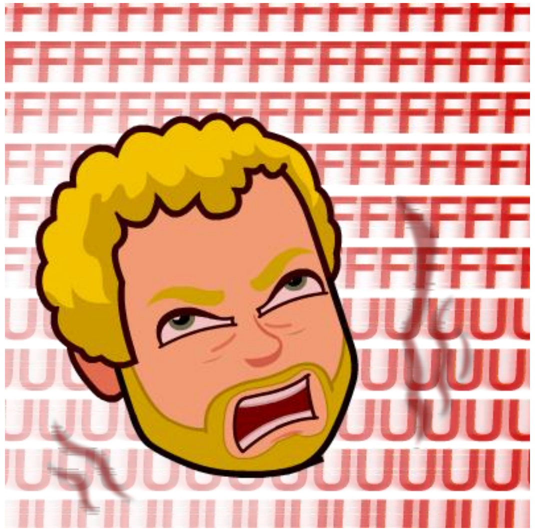
Why testing?

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

High-level view of workflow

```
Theorem foo :=
  forall x y …, p(x,y,…)
```





A better workflow

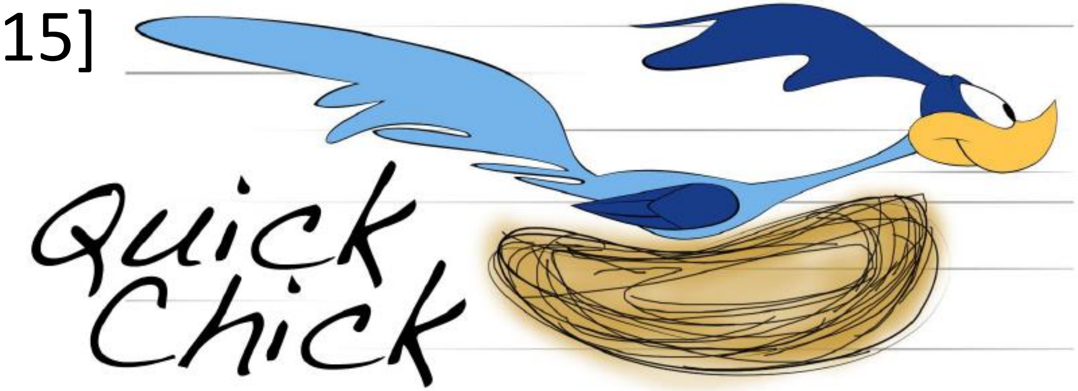
```
Theorem foo :=
  forall x y …, p(x,y,…)
```

Why testing?

- Supplemental to verification
- Already present in many proof assistants
  - Isabelle [Berghofer 2004, Bulwahn 2012]
  - Agda [Dybjer et al 2003]
  - ACL2 [Chamarthi et al 2011]
  - Not Coq!

Why testing?

- Supplemental to verification
- Already present in many proof assistants
  - Isabelle [Berghofer 2004, Bulwahn 2012]
  - Agda [Dybjer et al 2003]
  - ACL2 [Chamarthi et al 2011]
  - Not Coq!
- QuickChick [Paraskevopoulou et al 2015]
  - Coq port of Haskell QuickCheck
  - On steroids!

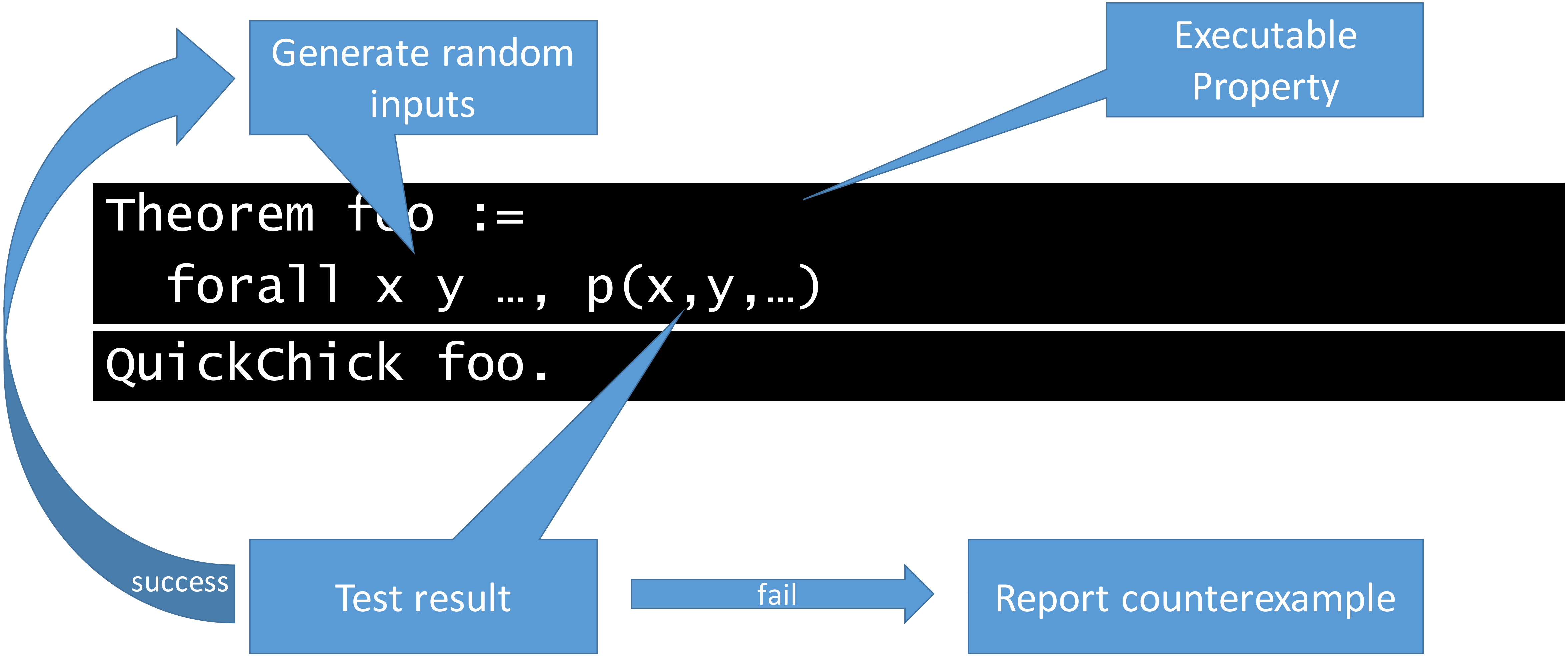


Overview of property-based testing

```
Theorem foo :=
  forall x y …, p(x,y,…)
```

```
QuickChick foo.
```

Overview of property-based testing



Generate random inputs

Executable Property

```
Theorem foo :=
  forall x y …, p(x,y,…)
QuickChick foo.
```

success

Test result

fail

Report counterexample

Overview

- Simple inductive types
- Random generation for simple inductive types
- The precondition problem
- Random generation for dependent inductive types

Running example : binary trees

Introduces tree type

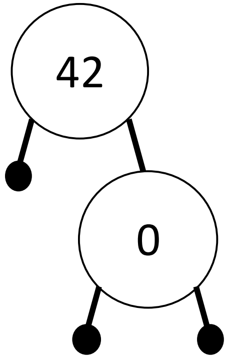
Leaves

Left and right subtrees

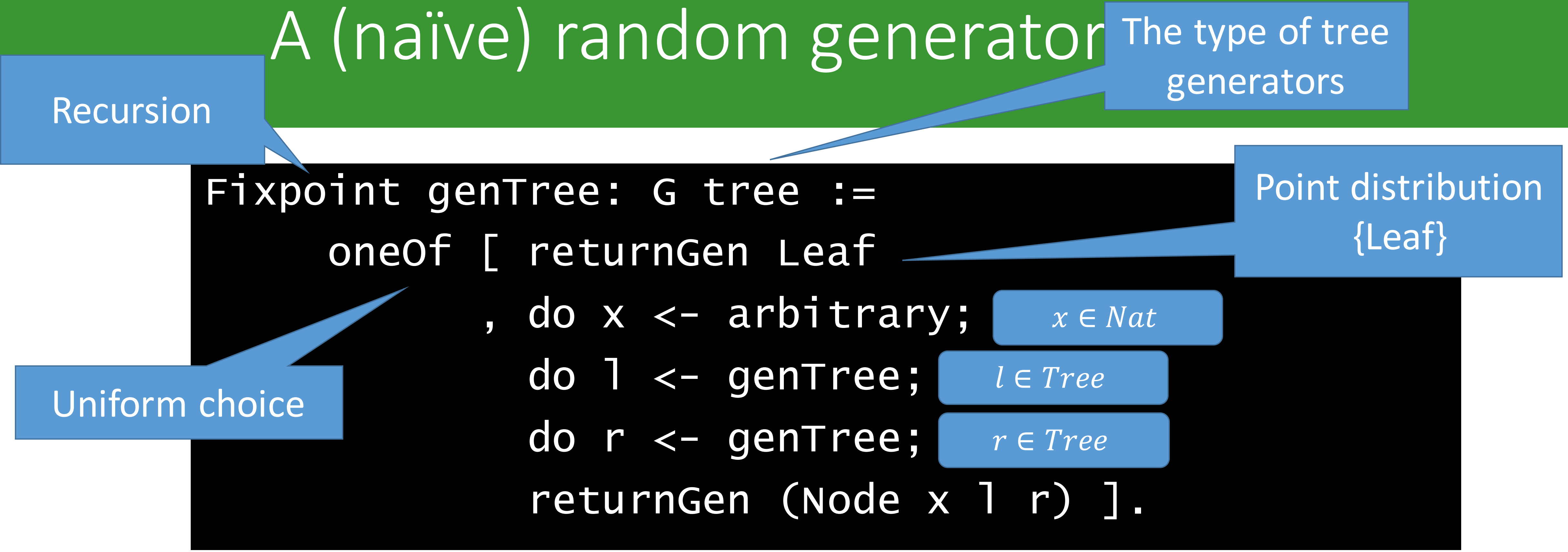
```
Inductive tree := Leaf : tree
                | Node : nat -> tree -> tree -> tree.
```

Integer Label

```
Node 42 Leaf (Node 0 Leaf Leaf)
```



A (naïve) random generator



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

Recursion

The type of tree generators

Point distribution {Leaf}

Uniform choice

$x \in Nat$

$l \in Tree$

$r \in Tree$

A (naïve) random generator

Recursion

The type of tree generators

```
Fixpoint genTree: G tree :=
     oneOf [ returnGen Leaf
           , do x <- arbitrary;
             do l <- genTree;
             do r <- genTree;
             returnGen (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
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 => returnGen 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 => returnGen Leaf
  | S size’ =>
    frequency [ (1, returnGen Leaf)
              , (size, do x <- arbitrary;
                       do l <- genTree size’
                       do r <- genTree size’
                       returnGen (Node x l r)) ].
```

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

if size = 0

if size = size’ + 1

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

$x \in Nat$

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

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

Recursive calls with smaller size

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

Distribution concerns

Well, what about uniform distributions?

- We could use Boltzmann samplers.
- But we usually do NOT want uniform distributions!
  - John’s talk tomorrow morning
  - Example: Finding bugs in the strictness analyzer of an optimizing compiler [Palka et al. 11]
    - Distribution heavily skewed towards terms containing “seq”

Properties with preconditions

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

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

If *x* is well typed

Then it is either a value or can take a step

Properties with preconditions

Generate x

If not, start over

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

Check $p(x)$

If check succeeds, test $q(x)$

mple condition: *complete* trees

`complete n t` denotes that `t` is a complete tree of height `n`

Type of logical propositions

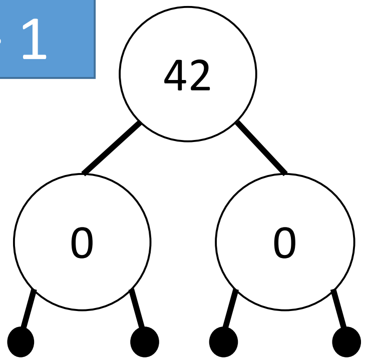
```
Inductive complete : nat -> tree -> Prop :=
  | c_leaf : complete 0 Leaf
  | c_node : forall n x l r,
              complete n l -> complete n r ->
              complete (S n) (Node x l r).
```

Leafs are complete trees of 0 height

If both l and r are complete trees of height n

S n = n + 1

Then we can combine them into a complete tree of size n + 1

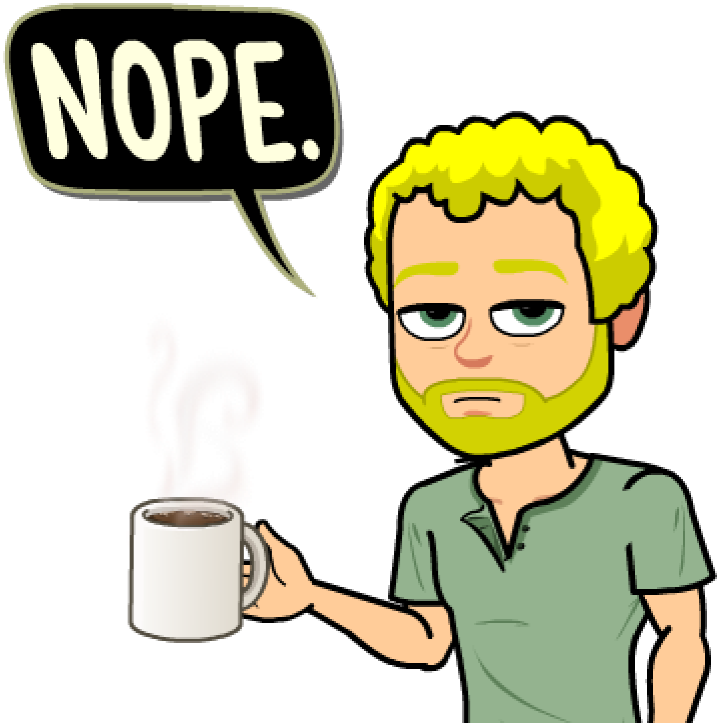


Let’s generate complete trees!

GOAL: Generate t, such that (`complete n t`) holds for a given n

Take 1 – Generate and test

- Assume we can *decide* whether a tree is complete
- Generate random trees
- Filter the complete ones



Take 2 – Custom generators

Solution: Write a generator that produces

All complete trees can be generated

Problem: Writing a Good Generator

All generated trees are complete

Distribution appropriate for testing

Custom generator for complete trees

This nat becomes input

```
Inductive complete : nat -> tree -> Prop :=
 | c_leaf : complete 0 Leaf
 | c_node : forall n x l r, complete n l -> complete n r ->
     complete (S n) (Node x l r).
```

```
Fixpoint genCTree (n : nat) : G tree :=
```

$\{t \mid complete\ n\ t\}$

Custom generator for complete trees

This nat becomes input

```
Inductive complete : nat -> tree -> Prop :=
  | c_leaf : complete 0 Leaf
  | c_node : forall n x l r, complete n l -> complete n r ->
      complete (S n) (Node x l r).
```

```
Fixpoint genCTree (n : nat) : G tree :=
  match n with
  | O => returnGen Leaf
  | S n’ =>
```

$\{t \mid complete\ n\ t\}$

No size (n determines size as well)

Custom generator for complete trees

This nat becomes input

```
Inductive complete : nat -> tree -> Prop :=
  | c_leaf : complete 0 Leaf
  | c_node : forall n x l r, complete n l -> complete n r ->
      complete (S n) (Node x l r).
```

```
Fixpoint genCTree (n : nat) : G tree :=
  match n with
  | O => returnGen Leaf
  | S n’ => do x <- arbitrary;
            do l <- genCTree n’;
            do r <- genCTree n’;
            returnGen (Node x l r) .
```

$\{t \mid complete\ n\ t\}$

$x \in Nat$

$l \in \{t \mid complete\ n'\ t\}$

$r \in \{t \mid complete\ n'\ t\}$

No size (n determines size as well)

Take 2 – Custom Generators

Write a generator that produces complete trees!

~~Problem: Writing a Good Generator~~

Take 2 – Custom Generators

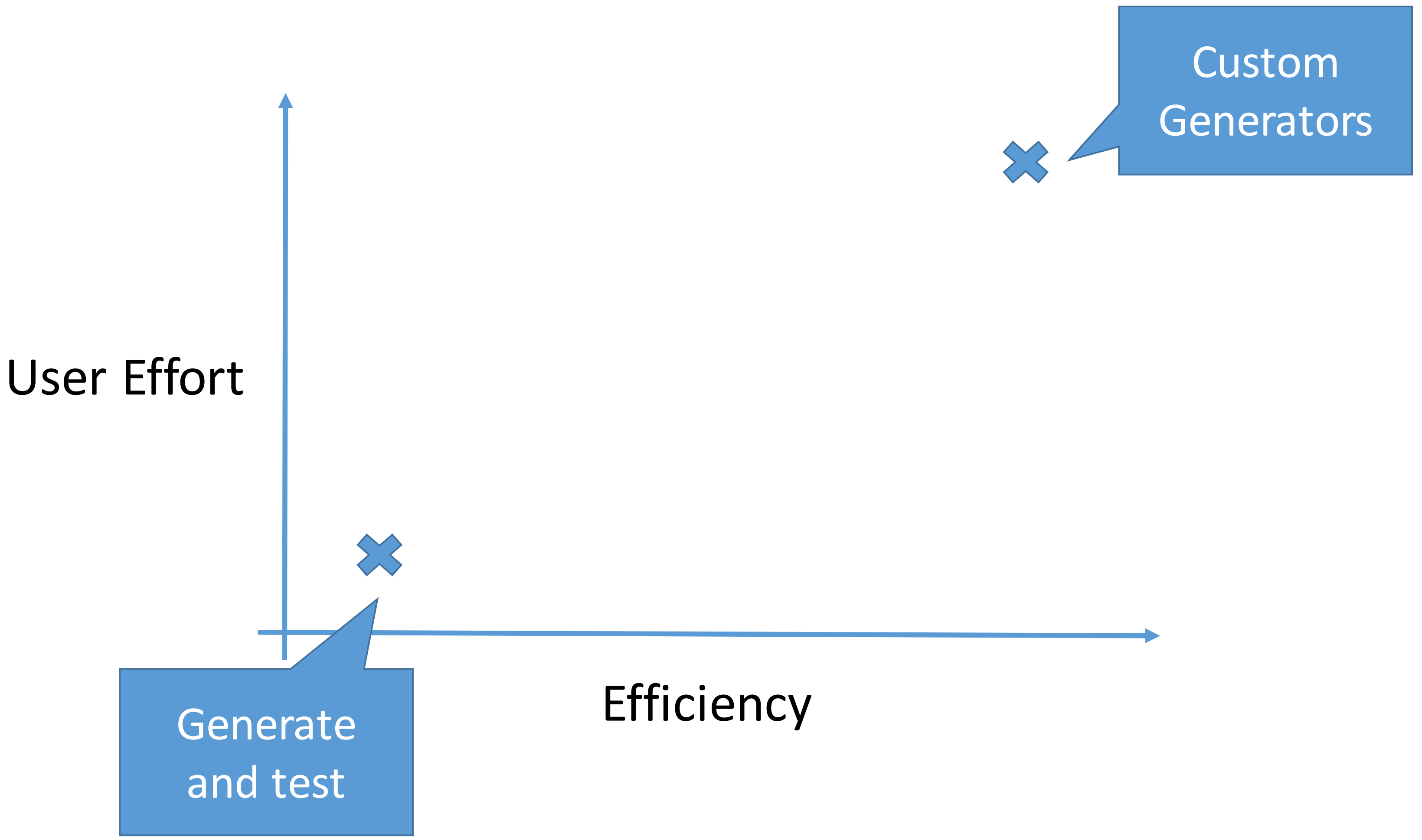
Write a generator that produces complete trees!

~~Problem: Writing a Good Generator~~

Problem: Too much boilerplate

Testing feedback should be immediate

Comparison


Custom Generators
User Effort
Generate and test
Efficiency

Take 3 - Narrowing

[Claessen et al. ’14, Fetscher et al. ’15, Lampropoulos et al. ’16]

- Borrows from functional logic programming
- Incremental generate and test
- Delay variable generation

```
Fixpoint isComplete (n : r            ee) :=
  match n with
  | O => match t with
         | Leaf => true
         | Node x l r => false
  | S n => match t with
           | Leaf => false
           | Node x l r => isComplete n’ l && isComplete n’ r
```

If n = 0, t must be Leaf

If n > 0, t must be a Node with complete subtrees

Take 3 - Narrowing

[Claessen et al. ’14, Fetscher et al. ’15, Lampropoulos et al. ’16]

rows fr ram

rement

- Delay variabl generation

```
Fixpoint isComplete (n : nat) (
  match n with
  | O => match t with
         | Leaf => true
         | Node x l r => false
  | S n’ => match t with
           | Leaf => false
           | Node x l r => isComplete n’ l && isComplete n’ r
```

Since n is fixed, only one branch can be taken

To the beginning? Too much wasted effort

n is input
t is to be generated
such that isComplete n t = true

To proceed we must instantiate the top constructor of t

If we pick Leaf, we’re done!

If not, we fail. Backtrack. But where?

Most recent choice!

Take 3 - Narrowing

[Claessen et al. ’14, Fetscher et al. ’15, Lampropoulos et al. ’16]

Since n is fixed, only one branch can be taken

rows from functional logic programm

remental generate and test

- Delay variable generation

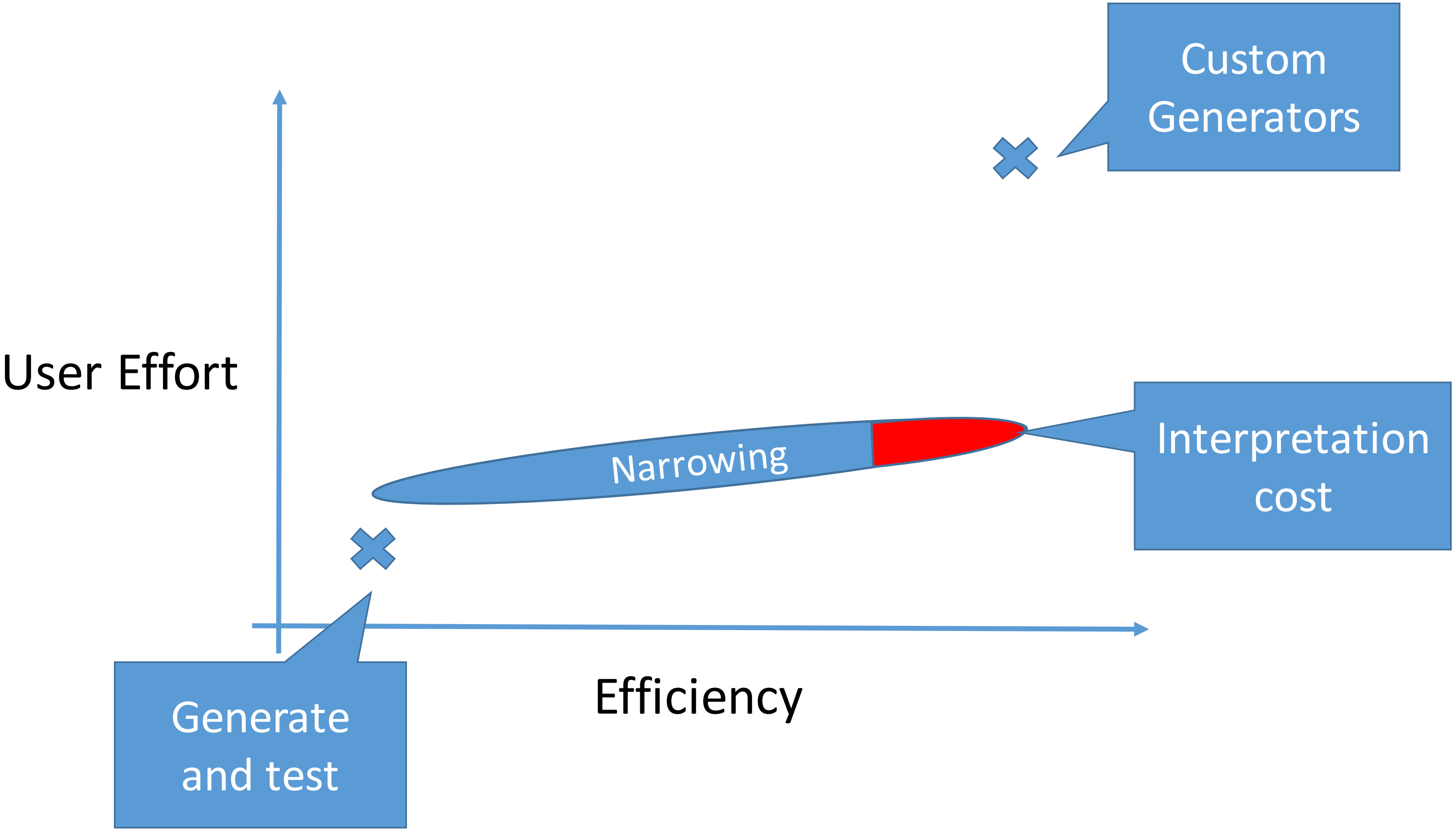
n is input
t is to be generated
such that isComplete n t = true

```
Fixpoint isComplete (n : nat) (t : tree) :=
  match n with
  | O => match t with
         | Leaf => true
         | Node x l r => false
  | S n => match t with
           | Leaf => false
           | Node x l r => isComplete n’ l && isComplete n’ r
```

If we pick Leaf, fail + backtrack

If Node, instantiate l + r recursively

Comparison



Custom Generators
User Effort
Narrowing
Interpretation cost
Generate and test
Efficiency

Our work

- Tackle preconditions in the form of dependent inductive types
- Produce generators that follow the narrowing approach

  (rather than writing an interpreter)

Rest of the talk

- High-level view of the generation algorithm via 3 examples
  - NonEmpty trees
  - Complete trees
  - Binary search trees
- Evaluation

Example 1 – nonEmpty

```
Inductive nonEmpty : tree -> Prop :=
  | ne : forall x l r, nonEmpty (Node x l r).
```

But how do we do that automatically?

```
Fixpoint genNonEmpty : G tree :=
  do x <- arbitrary;
  do l <- genTree;
  do r <- genTree;
  returnGen (Node x l r) .
```

$x \in Nat$

$l \in tree$

$r \in tree$

Example 1 – nonEmpty

Introduces unknown variable “t”

```
Inductive nonEmpty : tree -> Prop :=
  | ne : forall x l r, nonEmpty (Node x l r).
```

More unknowns

Unify “t” with “Node x l r”

```
Fixpoint genNonEmpty : G tree :=
  do x <- arbitrary;
  do l <- genTree;
  do r <- genTree;
  returnGen (Node x l r) .
```

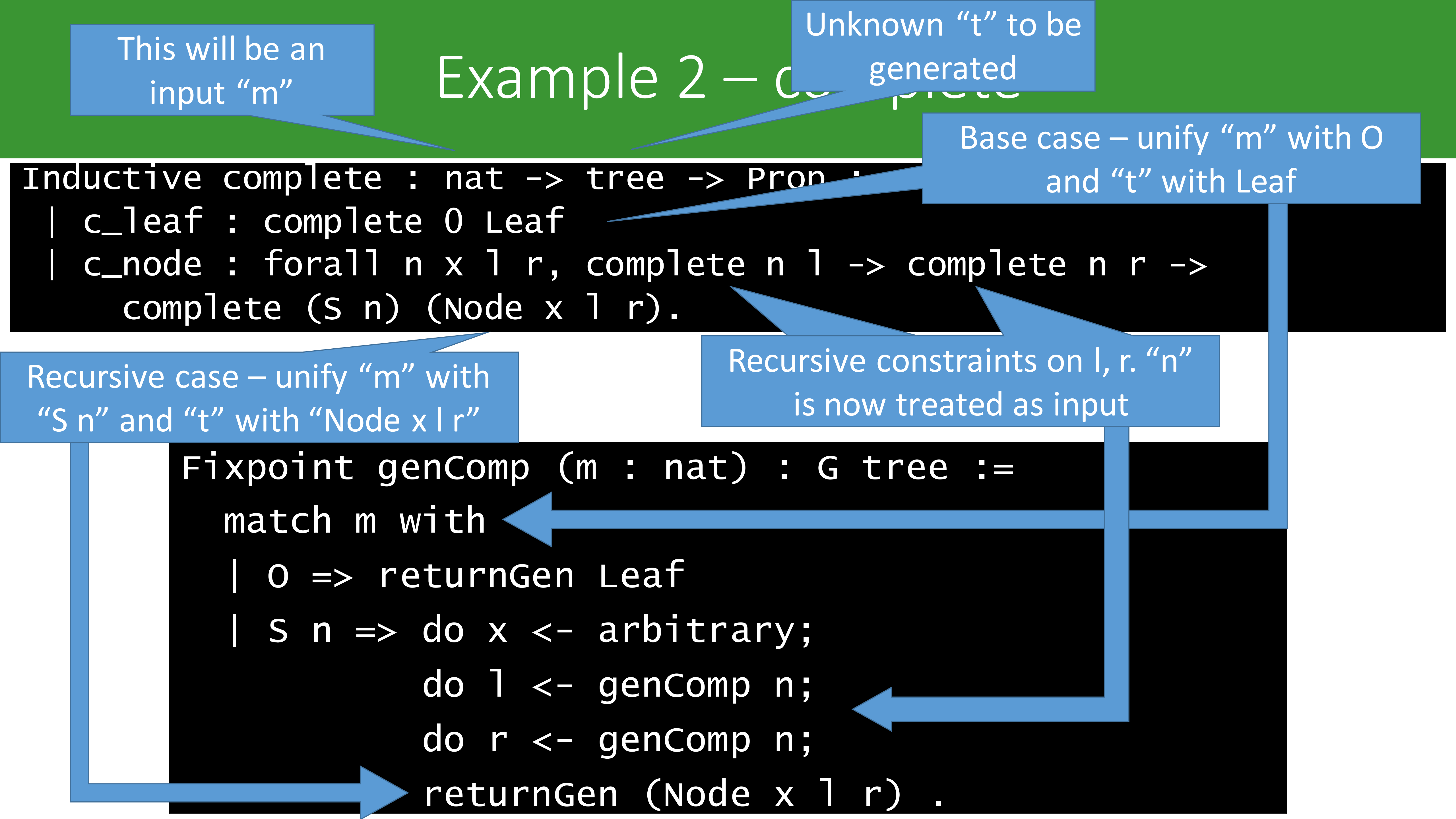
$x \in Nat$

$l \in tree$

$r \in tree$

x,l and r are unconstrained

Example 2 – complete



This will be an input “m”

Unknown “t” to be generated

Base case – unify “m” with O and “t” with Leaf

```
Inductive complete : nat -> tree -> Prop :=
  | c_leaf : complete O Leaf
  | c_node : forall n x l r, complete n l -> complete n r ->
      complete (S n) (Node x l r).
```

Recursive case – unify “m” with “S n” and “t” with “Node x l r”

Recursive constraints on l, r. “n” is now treated as input

```
Fixpoint genComp (m : nat) : G tree :=
  match m with
  | O => returnGen Leaf
  | S n => do x <- arbitrary;
           do l <- genComp n;
           do r <- genComp n;
           returnGen (Node x l r) .
```

mple 3 – Binary Search Trees

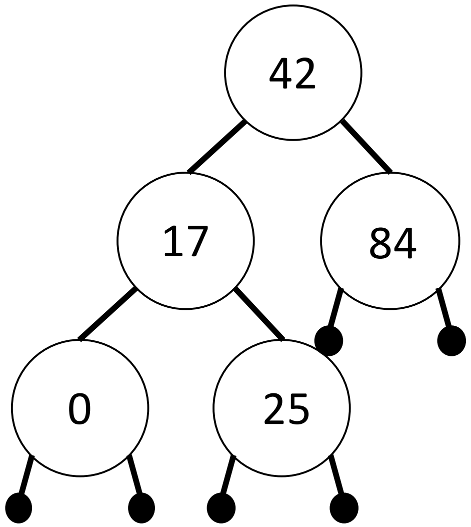
Binary search trees with elements between “lo” and “hi”

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

A Leaf is always a valid search tree

If lo < x < hi…

…and l,r are appropriate bsts



…then the combined Node is as well

These are inputs
“lo” and “hi”

ample 3 – Binary Search Trees

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O =>
  | S size’ =>
```

$\{t \mid bst\ lo\ hi\ t, size(t) \leq size\}$

Explicit size control

ample 3 – Binary Search Trees

These are inputs "lo" and "hi"

Unknown "t" to be generated

Base case – unify "t" with Leaf

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O =>
  | S size’ =>
```

These are inputs “lo” and “hi”

ample 3 – Binary ees

Unknown “t” to be generated

Base case – unify “t” with Leaf

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
```

Example 3 – Binary Search Trees

These are inputs “lo” and “hi”

Unknown “t” to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

Base case – unify “t” with Leaf

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1, … )]
```

Base case (bl)

Recursive case (bn)

These are inputs "lo" and "hi"

ample 3 – Binary ees

Unknown "t" to be generated

Recursive case – unify "t" with (Node x l r)

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl  : forall lo hi, bst lo hi Leaf
 | bn  : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size' =>
    frequency [(1, returnGen Leaf)
               (1,
```

ample 3 – Binary ees

These are inputs “lo” and “hi”

Unknown “t” to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

Recursive case – unify “t” with (Node x l r)

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1,




                                     returnGen (Node x l r)
                                      ) ].
```

ample 3 – Binary es

These are inputs “lo” and “hi”

Unknown “t” to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1,




                                  returnGen (Node x l r)
                                   ) ].
```

Generate x such that lo < x

Example 3 – Binary Search Trees

These are inputs "lo" and "hi"

Unknown "t" to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

Both x and hi are now fixed => Check

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size' =>
    frequency [(1, returnGen Leaf)
               (1, do x <- genGT lo;



                                   returnGen (Node x l r)
                                    ) ].
```

Generate x such that lo < x

$x \in \{lo + 1, \dots\}$

ample 3 – Binary es

These are inputs “lo” and “hi”

Unknown “t” to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

Both x and hi are now fixed => Check

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1, do x <- genGT lo;
                   if (x < hi)? then


                                 returnGen (Node x l r)
                   else           ) ].
```

Generate x such that lo < x

$x \in \{lo + 1, \ldots\}$

Example 3 – Binary Search Trees

These are inputs “lo” and “hi”

Unknown “t” to be generated

Recursively generate l and r

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl  : forall lo hi, bst lo hi Leaf
 | bn  : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1, do x <- genGT lo;
                   if (x < hi)? then


                                  returnGen (Node x l r)
                   else            ) ].
```

$x \in \{lo + 1, \ldots\}$

ample 3 – Binary Search Trees

These are inputs “lo” and “hi”

Unknown “t” to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

Recursively generate l and r

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1, do x <- genGT lo;
                   if (x < hi)? then do l <- genBst size’ lo x;
                                     do r <- genBst size’ x hi;
                                     returnGen (Node x l r)
                   else              ) ].
```

$x \in \{lo + 1, \ldots\}$

$l \in \{t \mid bst\ lo\ x\ t\}$

$r \in \{t \mid bst\ x\ hi\ t\}$

Example 3 – Binary Search Trees

These are inputs "lo" and "hi"

Unknown "t" to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G tree :=
  match size with
  | O => returnGen Leaf
  | S size’ =>
    frequency [(1, returnGen Leaf)
               (1, do x <- genGT lo;
                   if (x < hi)? then do l <- genBst size’ lo x;
                                     do r <- genBst size’ x hi;
                                     returnGen (Node x l r)
                   else  ???           ) ].
```

$x \in \{lo + 1, \ldots\}$

$l \in \{t \mid bst\ lo\ x\ t\}$

$r \in \{t \mid bst\ x\ hi\ t\}$

Example 3 – Binary Search Trees

These are inputs "lo" and "hi"

Unknown "t" to be generated

```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G (option tree) :=
  match size with
  | O => returnGen (Some Leaf)
  | S size' =>
    frequency [(1, returnGen (Some Leaf))
               (1, do x <- genGT lo;
                   if (x < hi)? then do l <- genBst size' lo x;
                                     do r <- genBst size' x hi;
                                     returnGen (Some (Node x l r))
                   else returnGen None) ].
```

Change to option types

Example 3 – Binary Search Trees

These are inputs “lo” and “hi”

Unknown “t” to be generated

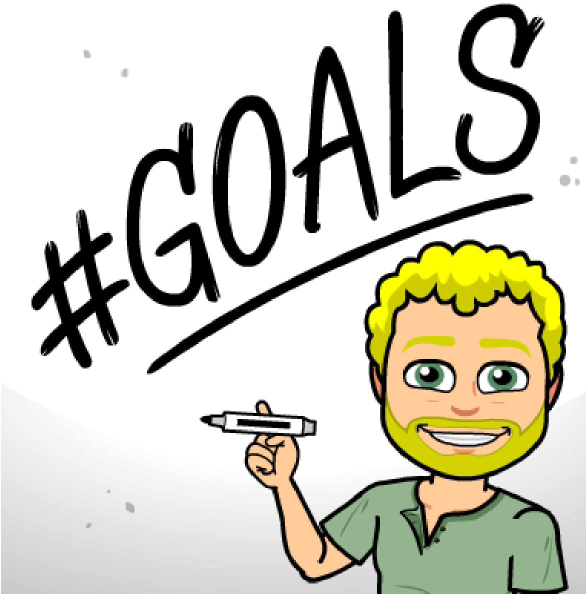
```
Inductive bst : nat -> nat -> tree -> Prop :=
 | bl : forall lo hi, bst lo hi Leaf
 | bn : forall lo hi x l r, lo < x -> x < hi ->
      bst lo x l -> bst x hi r -> bst lo hi (Node x l r).
```

```
Fixpoint genBst size lo hi : G (option tree) :=
  match size with
  | O => returnGen (Some Leaf)
  | S size’ =>
    backtrack [(1, returnGen (Some Leaf))
               (1, do x <- genGT lo;
                   if (x < hi)? then do l <- genBst size’ lo x;
                                     do r <- genBst size’ x hi;
                                     returnGen (Some (Node x l r))
                   else returnGen None) ].
```

Like frequency, but keeps trying other choices

Evaluation

- Use for testing past, current and future Coq projects
  - Software Foundations
  - Vellvm
  - GHC - Core



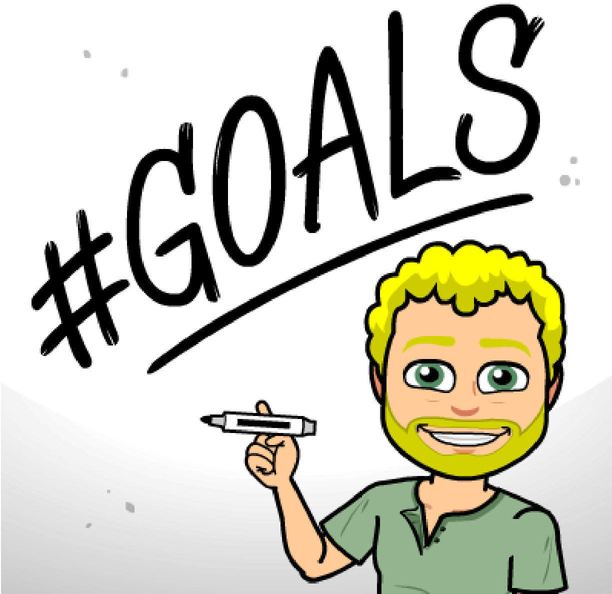
Evaluation

- Proof of correctness of the derived generators!
  - QuickChick framework provides support
  - Possibilistic correctness

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

All generated values satisfy p

All values that satisfy p can be generated



Thank you!