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]
Theorem foo :=
  \forall x, y \ldots, p(x, y, \ldots)
A better workflow

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

TESTING IN PROGRESS
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

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

• Simple inductive types
• Random generation for simple inductive types
• The precondition problem
• Random generation for dependent inductive types
Running example: binary trees

Inductive tree :=
| Leaf : tree

(Leaf 42)

Node 42 Leaf (Node 0 Leaf Leaf)
A (naïve) random generator for trees

Definition:
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
A (naïve) random generator for trees

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

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 trees

\textbf{Fixpoint} \texttt{genTree} (\texttt{size} : \texttt{nat}) : \texttt{G tree} :=

\{ t | \texttt{size}(t) \leq \texttt{size} \}

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

Fixpoint genTree (size : nat) : G tree :=
match size with
| 0 => returnGen Leaf
| S size' =>

size parameter : upper limit of the depth of the tree

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

Fixpoint genTree (size : nat) : G tree :=
match size with
| O => 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)) ].

size parameter : upper limit of the depth of the tree

\{ t | size(t) \leq size \}
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

∀x. p(x)

∀x. p(x) → q(x)

If x is well typed

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

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

- Generate $x$
- Check $p(x)$
- If check succeeds, test $q(x)$
- If not, start over
A simple condition: \textit{complete} trees

\texttt{type complete} \ n \ t \ \texttt{denotes that} \ t \ \texttt{is a complete tree of height} \ n

\texttt{inductive complete : nat -> tree -> Prop :=}

\begin{align*}
| & \texttt{c_leaf : complete 0 Leaf} \\
| & \texttt{c_node : forall n x l r, complete n l -> complete n r -> complete (S n) (Node x l r).}
\end{align*}

Type of logical propositions

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 \((\text{complete } n \ t)\) 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 complete trees!

Problem: Writing a Good Generator

- All generated trees are complete
- All complete trees can be generated
- Distribution appropriate for testing
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 | complete n t}
Custom generator for complete trees

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' =>

This nat becomes input

No size (n determines size as well)
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
  | 0 => returnGen Leaf
  | S n' => do x <- arbitrary;
            do l <- genCTree n';
              do r <- genCTree n';
              returnGen (Node x l r) .

This nat becomes input

No size (n determines size as well)
Write a generator that produces complete trees!

Problem: Writing a Good Generator
Write a generator that produces complete trees!

Problem: Writing a Good Generator

Problem: Too much boilerplate

Testing feedback should be immediate
Comparison

- User Effort
- Generate and test
- Efficiency
- Custom Generators
• Borrows from functional logic programming
• Incremental generate and test
• Delay variable generation

Fixpoint isComplete (n : nat) : Tree :=
match n with
| 0 => 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


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

Fixpoint isComplete (n : nat) (t : tree) :=

match n with
| 0 => 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
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!

n is input
T is to be generated such that isComplete n t = true
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 : nat) (t : tree) :=
  match n with
  | 0 => 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
n is input
t is to be generated such that isComplete n t = true
If we pick Leaf, fail + backtrack
If Node, instantiate l + r recursively
Comparison

User Effort

Generate and test

Efficiency

Interpretation cost

Custom Generators

Narrowing
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; \( x \in \text{Nat} \)
do l <- genTree; \( l \in \text{tree} \)
do r <- genTree; \( r \in \text{tree} \)
returnGen (Node x l r) .
Example 1 – nonEmpty

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

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

Introduces unknown variable “t”
More unknowns
Unify “t” with “Node x l r”

x ∈ Nat
l ∈ tree
r ∈ tree

x, l and r are unconstrained
Example 2 – complete

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).

This will be an input “m”

Unknown “t” to be generated

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

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) .

This will be an input “m”

Unknown “t” to be generated

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

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
Example 3 – Binary Search Trees

A Leaf is always a valid search tree

If \( lo < x < hi \)…

…then the combined Node is as well

…and \( l,r \) are appropriate bst's

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

Inductive \( \text{bst} : \text{nat} \rightarrow \text{nat} \rightarrow \text{tree} \rightarrow \text{Prop} \):

\[
\begin{align*}
| \text{bl} : & \forall lo \ hi, \ \text{bst} \ lo \ hi \ \text{Leaf} \\
| \text{bn} : & \forall lo \ hi \ x \ l \ r, \ lo < x \rightarrow x < hi \rightarrow \\
& \ \text{bst} \ lo \ x \ l \rightarrow \text{bst} \ x \ hi \ r \rightarrow \text{bst} \ lo \ hi \ (\text{Node} \ x \ l \ r).
\end{align*}
\]
Example 3 – Binary Search Trees

Fixpoint genBst size lo hi : G tree :=
    match size with
    | 0 =>
    | S size’ =>

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).

These are inputs “lo” and “hi”

Explicit size control
Example 3 – Binary Search Trees

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

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).

Unknown “t” to be generated
Base case – unify “t” with Leaf

These are inputs “lo” and “hi”
Example 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 => returnGen Leaf
  | S size' =>

These are inputs “lo” and “hi”
Unknown “t” to be generated
Base case – unify “t” with Leaf
Example 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 => returnGen Leaf
| S size' =>
  frequency [(1, returnGen Leaf)
            (1, ... )]

Example 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 => returnGen Leaf
  | S size' =>
    frequency [(1, returnGen Leaf)]
    (1,
Example 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 => returnGen Leaf
  | S size' =>
    frequency [(1, returnGen Leaf)
                 (1, returnGen (Node x l r))].

Unknown “t” to be generated
Recursive case – unify “t” with (Node x l r)
These are inputs “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 r -> bst x hi r -> bst lo hi (Node x l r).

Fixpoint genBst size lo hi : G tree :=
  match size with
  | 0 => returnGen Leaf
  | S size' =>
    frequency [(1, returnGen Leaf)
                (1, returnGen (Node x l r))].
Example 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
| 0 => returnGen Leaf
| S size' =>
  frequency [(1, returnGen Leaf)
  (1, do x <- genGT lo;
    x ∈ {lo + 1,...} returnGen (Node x l r)
  )].

Both x and hi are now fixed => Check

Generate x such that lo < x
Unknown “t” to be generated

These are inputs “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).

Fixpoint genBst size lo hi : G tree :=
match size with
| 0 => returnGen Leaf
| S size' =>
  frequency [(1, returnGen Leaf)
    (1, do x <- genGT lo;
       if (x < hi)? then
         returnGen (Node x l r)
       else

Unknown “t” to be generated

These are inputs “lo” and “hi”

Generate x such that lo < x

Both x and hi are now fixed => Check

x \in \{lo + 1, \ldots\}
Example 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
  | 0 => returnGen Leaf
  | S size' =>
    frequency [(1, returnGen Leaf)]
    (1, do x <- genGT lo; \x ∈ \{lo + 1, ...\}
    if (x < hi)? then
      returnGen (Node x l r)
    else
      )].

Unknown “t” to be generated
Recursively generate l and r
These are inputs “lo” and “hi”
Example 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
  | 0 => 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 ]

Unknown “t” to be generated
Recursively generate l and r
These are inputs “lo” and “hi”
Example 3 – Binary Search Trees

Fixpoint genBst size lo hi : G tree :=
match size with
| O => returnGen Leaf
| S size' =>
  frequency [(1, returnGen Leaf) (~ x ∈ \{lo + 1, \ldots\})
    (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 ??? )].

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).

Unknown “t” to be generated

These are inputs “lo” and “hi”
Example 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 (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) ].

Unknown “t” to be generated

These are inputs “lo” and “hi”

Change to option types
Example 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 (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) ].

These are inputs “lo” and “hi”
Unknown “t” to be generated
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!