Generating Good Generators for Inductive Relations

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

Property-based random testing (PBRT) is widely used in the functional programming and verification communities. For testing simple properties, PBRT tools such as QuickCheck can automatically generate random inputs of a given type. But for more complex properties, effective testing often demands generators for random inputs that belong to a given type *and* satisfy some logical condition. QuickCheck provides a library of combinators for building such generators by hand, but this can be tedious for simple conditions and error prone for more complex ones. Fortunately, the process can often be automated. The most prominent method, *narrowing*, works by traversing the structure of the condition, lazily instantiating parts of the data structure as constraints involving them are met.

We show how to use ideas from narrowing to compile a large subclass of Coq's inductive relations into efficient generators, avoiding the interpretive overhead of previous implementations. More importantly, the same compilation technique allows us to produce proof terms certifying that each derived generator is *good*—i.e., sound and complete with respect to the inductive relation it was derived from. We implement our algorithm as an extension of QuickChick, an existing tool for property-based testing in Coq. We evaluate our method by automatically deriving good generators for the majority of the specifications in *Software Foundations*, a formalized textbook on programming language foundations.

CCS Concepts: •**Software and its engineering → General programming languages;**

Additional Key Words and Phrases: Random Testing, Property-based Testing, Coq, QuickCheck, QuickChick, Narrowing

ACM Reference format:
Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 28 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Property-based random testing (PBRT) is a popular technique for quickly discovering software errors. Starting with Haskell's QuickCheck (Claessen and Hughes 2000), property-based testing tools have spread to a wide variety of languages (Arts et al. 2008; Hughes 2007; Lindblad 2007; Pacheco and Ernst 2007; Papadakis and Sagonas 2011). The benefits of PBRT are also enjoyed by users of automated theorem provers like ACL2 (Chamarthi et al. 2011) and proof assistants like Isabelle (Bulwahn 2012a), Agda (Dybjer et al. 2004), and, more recently, Coq (Paraskevopoulou et al. 2015); testing in these settings can save wasting time and effort on false conjectures (Dybjer et al. 2003).

For complex properties, setting up PBRT-style testing can involve substantial work. Particular effort is required for specifications involving *sparse preconditions*: ones that hold for only a small fraction of the input space. For example, consider the following property (written in Coq's internal functional language, Gallina), which states that inserting an element into a sorted list preserves sortedness:

```
Definition prop_insert (x : nat) (l : list nat) := sorted l ==> sorted (insert x l).
```

If we test this property by generating random lists, throwing away ones that are not sorted, and checking the conclusion `sorted (insert x l)` for the rest—the *generate-and-test* approach—we will waste most of our time

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

generating and discarding unsorted lists; worse, the distribution of the lists that we do *not* discard will be strongly skewed toward short ones, which might fail to expose bugs that only show up for larger inputs.

To tackle properties with sparse preconditions, QuickCheck provides a comprehensive library of combinators for writing *custom generators* for well-distributed random values. Such generators are heavily—and successfully—employed by QuickCheck users. However, writing them can be both complex and time consuming, sometimes to the point of being a research contribution in its own right (Hriţcu et al. 2013, 2016; Palka et al. 2011)!

This has led to interest in automated techniques for enumerating or randomly generating data structures satisfying some desired condition (Bulwahn 2012b; Claessen et al. 2014; Fetscher et al. 2015; Gligoric et al. 2010; Kuraj and Kuncak 2014; Lampropoulos et al. 2017). One particularly successful technique is *narrowing* (Antoy 2000), a concept borrowed from functional logic programming. The idea of narrowing is to construct a random data structure lazily while traversing the definition of the predicate it must satisfy. For example, consider the `sorted` predicate: a list with at most one element is always sorted, while a list with at least two elements `(x::y::ys)` is sorted if `x` is smaller than `y` and `(y::ys)` is itself sorted.[1]

```
Fixpoint sorted (l : list nat) :=
  match l with
    | [] => true
    | [x] => true
    | x::y::ys => x <=? y && sorted (y::ys)
  end.
```

To generate a list `l` satisfying `sorted l` using narrowing, we look first at the pattern match and choose randomly whether to instantiate `l` to an empty list, a list of one element, or a list with at least two elements. In the first case, we have a value satisfying the predicate and we are done. In the second, we have a free choice for the value of `x` and then we are done. In the third case, we next encounter the constraint `x <=? y`; we generate values for `x` and `y` to satisfy this constraint locally and then proceed recursively to generate a value for `ys` satisfying `sorted (y::ys)`. The next time we explore `sorted`, the parameter `l` is partially instantiated—it consists of the known value `y` consed onto the unknown value `ys`; this means that we cannot choose the empty branch; if we choose the second branch we do not have to generate `x`; and if we choose the third branch we only have to generate the second element of the list (to be bigger than the first) and proceed recursively. Automatic narrowing-based generators can achieve testing effectiveness (measured as bugs found per test generated) comparable to hand-written custom generators, even for challenging examples (Claessen et al. 2014; Fetscher et al. 2015; Lampropoulos et al. 2017).

Unfortunately, both hand-written and narrowing-based automatic generators are subject to bugs. For hand-written ones, this is because generators for complex conditions can often be complex, often more than the condition itself; moreover, they must be kept in sync if the condition is changed, another source of bugs. Automatic generators do not suffer from the latter problem, but narrowing solvers are themselves rather complex beasts, whose correctness is therefore questionable. (The tool of Lampropoulos et al. (2017) does come with a proof, but only for an abstract model of part of the core algorithm, not to the full implementation.)

Bugs in generators can come in two forms: they can generate too much, or too little—i.e., they can be either unsound or incomplete. Unsoundness can lead to false positives, which can waste significant amounts of time. Incompleteness can lead to ineffective testing, where certain bugs in the program under test can never be found because the generator will never produce an input that provokes them. Both problems can be detected—unsoundness by double-checking whether generated values satisfy the property, incompleteness by techniques such as mutation testing (Jia and Harman 2011)—and unsoundness can be mitigated by filtering away generated values that fail the double-check, but incompleteness bugs can require substantial effort to understand and repair.

[1]Strictly speaking, this definition is not legal in Gallina, since `y::ys` is not recognized as a strict subterm of `l`. Expert Coq readers will know how it can be massaged to make the termination checker happy; others can ignore this detail.

The core contribution of this paper is a method for compiling a large class of logical conditions, expressed as Coq inductive relations, into random generators together with soundness and completeness proofs for these generators. (We do *not* prove that the compiler itself is correct in the sense that it can only produce good generators; rather, we adopt a *translation validation* approach (Pnueli et al. 1998) where we produce a checkable certificate of correctness along with each generator.) A side benefit of this approach is that, by compiling inductive relations into generators, we avoid the interpretive overhead of existing narrowing-based generators. As discussed by Lampropoulos et al. (2017), this overhead is one of the reasons existing generators can be an order of magnitude slower than their hand-written counterparts.

We have implemented our method as an extension of QuickChick, a QuickCheck variant for PBRT in Coq (Paraskevopoulou et al. 2015). Using QuickChick, a Coq user can write down desired properties like

```
Conjecture preservation : forall (t t' : tm) (T : ty),
  |- t \in T  ->  t ===> t'  ->  |- t' \in T.
```

and look for counterexamples with no additional effort:

```
 QuickChick preservation.    ⟶     QuickChecking preservation... Passed 10000 tests
```

Our technical contributions are as follows:

- We present a narrowing-inspired method for compiling a large class of inductive definitions into random generators. Section 3 introduces our compilation algorithm through a sequence of progressively more complex examples; Section 4 describes it in full detail.
- We show how this algorithm can also be used to produce proofs of (possibilistic) correctness for every derived generator (Section 5). Indeed, by judicious application of Coq's typeclass features, we can use exactly the same code to produce both generators and proof terms.
- We implement the algorithm as an extension of QuickChick, further integrating testing and proving in the Coq proof assistant and providing more push-button-style automation while retaining customizability (Section 6).
- To evaluate the applicability of our method, we applied the QuickChick implementation to a large part of *Software Foundations* (Pierce et al. 2016), a machine-checked textbook on programming language theory. Of the 232 nontrivial theorems we considered, 84% are directly amenable to PBRT (the rest are higher-order properties that would at least require significant creativity to validate by random testing); of these, 83% can be tested using our algorithm. We discuss these findings in detail in Section 7.1.
- To evaluate the efficiency of our generators, we compare them to fine-tuned handwritten generators for information-flow control abstract machines (Section 7.2). The derived generators were 1.75× slower than the custom ones, demonstrating a significant speedup over previous interpreted approaches such as Luck (Lampropoulos et al. 2017).

Section 2 introduces QuickChick and provides necessary background and notations for the rest of the paper. Section 8 discusses related work. We conclude and draw directions for future work in Section 9.

2 QUICKCHICK : QUICKCHECK IN COQ

We begin with an overview of property-based random testing within Coq using QuickChick (Dénès et al. 2014), introducing its basic notations and typeclasses. Our running example for this section and the rest of the paper will be the following data type of binary Trees of natural numbers:

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

Consider the following simple function that completely mirrors its input tree, that is it interchanges the left and right children of all Nodes:

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

A natural property we expect to hold is that `mirror` is an *involution*: mirroring a tree twice yields the original tree. Armed with an equality on trees we can easily state this property in Coq:

```
Definition mirror_involution (t : Tree) : bool  :=  mirror (mirror t) == t.
```

In order to test the property in QuickChick, we require three things: a generator for random data, a printer for counterexamples, and a shrinker for finding minimal counterexamples. In this paper we focus on generators; see Claessen and Hughes (2000) for discussion of the others.

2.1 Generators

Just like QuickCheck, QuickChick provides a variety of generator combinators that facilitate the generation of complex data structures. For example, we can encode a simple, naive generator for binary trees that flips a coin to decide whether to create a `Leaf` or a `Node`, and proceeds recursively if necessary.

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

The `oneOf` combinator serves the role of the coin flip; it takes a list of generators and picks one of them at random. Here we have two generators: the first generator always returns a `Leaf` in the generator monad `G`, described below; the second one produces an `arbitrary` number and the left and right subtrees `l` and `r` recursively, before combining the generated data into a `Node`. We use the monadic `do!` notation provided by QuickChick to sample the different variables in sequence.

Unfortunately, the `genTree` generator as written does not actually work: there is no guarantee that it terminates; indeed, the expected size of generated trees is infinite! In the random testing community, to overcome this limitation, it is common to introduce a `size` parameter to limit the size of generated terms. In the following second attempt at a generator, we use a natural number `size` to signify the maximum depth of the generated tree. In the rest of the paper we will refer to size parametric generators as *bounded* generators.

```
Fixpoint genTreeSized (size : nat) :=
  match size with
  | O => ret Leaf
  | S size' => freq [ (1, ret Leaf)
                    ; (size, do! x <- arbitrary;
                             do! l <- genTreeSized size';
                             do! r <- genTreeSized size';
                             ret (Node x l r)) ]
  end.
```

In the above bounded generator, when the size is zero, the only tree that can be generated is a `Leaf`. When the size is nonzero, we have a choice: either generate a `Leaf`, or generate a `Node` where the left and right subtrees `l` and r are generated recursively with size decremented by 1. This choice is made using the freq combinator (short for frequency); freq takes a list of *weighted* generators and picks one of them, based on the induced discrete distribution. For example, gen_tree_sized creates a Leaf $\frac{1}{\texttt{size}+1}$ of the time and a Node $\frac{\texttt{size}}{\texttt{size}+1}$ of the time. The freq combinator gives the user a degree of local distribution control that can be used to fine-tune the distribution of generated data, a crucial feature in practice.

Once we have a bounded generator we can obtain an *unbounded* one using the sized combinator. To understand what sized does we must first peek at the definition of the G monad.

```
Definition G (A:Type) : Type := nat -> RandomSeed -> A.
```

G is represented as a "reader" monad with two parameters, a random seed and a size parameter. When QuickChick runs a computation in the G monad to generate random elements, it will use increasingly larger size parameters until either a counterexample is found or a predefined size limit is reached. Given a bounded generator, sized will apply it to the size parameter that is internal to the representation of the generator, making it implicitly bounded.

```
Definition sized {A : Type} (f : nat -> G A) : G A := fun n r => (f n) n r.
```

2.2 QuickChick Typeclasses

Generation. The binary trees of this section always contain nats as labels. Most of the time, however, Coq users would use a polymorphic tree data type instead. Revisiting the bounded Tree generator in the polymorphic case, we would need to generate an arbitrary x of the label type. To avoid cluttering the definitions and calls to genTreeSized with specific generators, we leverage Coq's *typeclasses* (Sozeau and Oury 2008; Wadler and Blott 1989).

Specifically, just like Haskell's QuickCheck, QuickChick provides the Gen typeclass with a single method, arbitrary, that produces random elements of the underlying type A.

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

Looking back at genTreeSized, the arbitrary method we used to generate x comes from the default Gen instance for natural numbers.

In addition, unlike Haskell's QuickCheck, QuickChick introduces another class, GenSized: the class of bounded generators that explicitly depend on a size parameter.

```
Class GenSized (A : Type) := { arbitrarySized : nat -> G A }.
```

QuickChick automatically converts GenSized instances to obtain corresponding Gen instances by applying sized to the bounded generator of the GenSized class. This is done by adding the following instance

```
Instance GenOfGenSized {A} {H : GenSized A} : Gen A := {| arbitrary := sized arbitrarySized |}.
```

The above instance provides an arbitrary method by applying sized to the arbitrarySized method of the GenSized instance. We will leverage this relation between GenSized and Gen during proof generation (Section 5).

Just like in Haskell's QuickCheck ecosystem, it is straightforward to automatically derive GenSized instances (as well as instances of similar typeclasses for shrinking and printing) (Xia 2017). Using the techniques that will be described in Section 5, we can in addition provide proofs of correctness of the derived generators.

Decidability. In the context of a proof assistant like Coq, we are faced with a challenge that is non-existent in a functional setting: non-executable specifications. Consider, for example, a different formulation of the mirror_involution property that uses syntactic equality instead.

```
Definition mirror_involution (t : Tree) : Prop := mirror (mirror t) = t.
```

Since the conclusion of this property is in `Prop`, we cannot actually execute it for an arbitrary generated tree `t` to decide whether it holds or not. To that end, QuickChick provides a `Dec` typeclass over `Prop`s `P`, with a single method `dec` that provides a proof of either `P` or its negation.

```
Class Dec (P : Prop) : Type := { dec : {P} + {~ P} }.
```

A predicate that is an instance of this class can be used as the conclusion of any property, which is automatically testable via use of typeclass resolution. In addition, a user can write `P?` to explicitly convert a property to a boolean value.

Constrained Generation. Just like `arbitrarySized` and `arbitrary` are the methods of the `GenSized` and `Gen` typeclasses for simple inductive types, `arbSizedST` and `arbST` are methods of `GenSizedSuchThat` and `GenSuchThat` classes for constrained generation.

```
Class GenSizedSuchThat (A : Type) (P : A -> Prop) :=
  { arbitrarySizeST : nat -> G (option A) }.

Class GenSuchThat (A : Type) (P : A -> Prop) :=
  { arbitraryST : G (option A) }.
```

Given an inductive predicate `P` over some type `A`, these typeclasses provide access to bounded and unbounded generators for elements of `A` that claim to satisfy `P` directly. However, since in general there may be no such such elements (we will see an example in the next section) the result type of these methods is an `option`. Again, once we have an instance of `GenSizedSuchThat` we can obtain automatically an instance of `GenSuchThat` by adding the appropriate instance.

To chain different monadic option actions we could use a bind like what we would get if we were using monad transformers for the `G` and `option` monads:

```
Definition bindGenOpt' {A B} (m : G (option A)) (k : A -> G (option B)) :=
  do! a <- m;
  match a with
  | Some a' => k a'
  | None => ret None
  end.
```

However, we can specialize the bind to achieve more control over local backtracking, which can lead to great efficiency gains in practice (Claessen et al. 2014). Instead of failing when the generator fails, we can try again, until we reach a pre-specified number of times; here, the implicit size parameter of the `G` monad.

```
Definition bindGenOpt {A B} (m : G (option A)) (k : A -> G (option B)) :=
  let fix aux lim :=
      match lim with
      | O => ret None
      | S lim' => do! a <- m;
                  match a with
                  | Some a' => k a'
                  | None => aux lim'
                  end
      end in
  sized aux.
```

We use `doM!` notation instead of the more common `do!` notation to chain actions in this way.

3 GOOD GENERATORS, BY EXAMPLE

While automating generation for simple types is a much-needed addition to QuickChick, it still doesn't suffice for testing a common type of interesting properties: properties with preconditions. The main focus of this paper is to derive *correct* generators for simply-typed data satisfying dependently-typed, inductive invariants. This section uses examples to showcase different behaviors that our generation algorithm needs to exhibit; the algorithm itself will be described more formally in the following section. In particular, we are going to give a few progressively more complex inductive characterizations of trees, and detail how we can automatically produce a generator for trees satisfying those characterizations.

Nonempty trees. Our first example is nonempty trees, i.e., trees that are not just leaves.

```
Inductive nonempty : Tree -> Prop :=
| NonEmpty : forall x l r, nonempty (Node x l r).
```

From a user's perspective, we can quickly come up with a generator for nonempty trees: we just need to create arbitrary `x`, `l` and `r` and combine them into a `Node`.

```
Definition gen_nonempty : G (option Tree) :=
  do! x <- arbitrary;
  do! l <- arbitrary;
  do! r <- arbitrary;
  ret (Some (Node x l r)).
```

But how could we automate this process?

We know that we want to generate a tree `t` satisfying `nonempty`; that means that we need to pick some constructor of `nonempty` to satisfy. Since there is only one constructor, we only have one option, `NonEmpty`. By looking at the conclusion of the `NonEmpty` constructor we know that `t` must be a `Node`. This can be described by a *unification* procedure. Specifically, we introduce an *unknown variable* `t` (similar to logical variables in logic programming, or unification variables in type inference) plus one unknown variable for each universally quantified variable of the constructor (here `x`, `l` and `r`). We then proceed to unify `t` with `(Node x l r)`. Since there are no more constraints—we call them "hypotheses"—in `NonEmpty`, and since `x`, `l` and `r` are still completely unknown, we instantiate them arbitrarily (using the `Gen` instance for natural numbers that is provided by default, as well as the instance for `Trees` derived automatically in the previous section).

Complete Trees. For our second example of a condition, consider complete trees (also known as perfect trees): binary trees whose leaves are all at the same depth. The shape of a complete tree can be fully characterized by its depth: a complete tree of depth zero is necessarily a `Leaf`, while a complete tree of depth `n+1` is formed by combining two complete trees of depth `n` into a `Node`. This is reflected in the following inductive definition:

```
Inductive complete : nat -> Tree -> Prop :=
| CompleteLeaf :
    complete 0 Leaf
| CompleteNode : forall n x l r,
    complete n l -> complete n r ->
    complete (S n) (Node x l r).
```

Since `complete` has two parameters, we need to decide whether the derived generator produces all of them or treats some of them as inputs, i.e., we need to assign *modes* to the parameters, in the sense of functional logic programming. Let's assume that that first parameter is an input to the generator (called `in1`), and we want to generate trees `t` that satisfy `complete in1 t`. Once again, we introduce an unknown variable `t` (that we want to generate), as well as an unknown variable for `in1`: since the generator will receives `in1` as an argument, we don't know its actual value at derivation time!

We now have two constructors to choose from to try to satisfy, `CompleteLeaf` and `CompleteNode`. If we pick `CompleteLeaf` we need to unify `t` with `Leaf` and `in1` with `O`. Since `t` is unconstrained at this point, we can always unify it with `Leaf`. By contrast since we don't know the value of `in1` at derivation time, we need to produce a *pattern match* on it: if `in1` is `O`, then we can proceed to return a `Leaf`, otherwise we can't satisfy `CompleteLeaf`.

On the other hand, if we pick `CompleteNode`, we introduce new unknowns `n`, `x`, `l` and `r` for the universally quantified variables. We proceed to unify `t` with `Node x l r` and `m` with `S n`. Like before, we need to pattern match on `in1` to decide at runtime if it is nonzero; we bind `n` in the pattern match and treat it as an input from that point onward. We then handle the recursive constraints on `l` and `r`, instantiating both the left and right subtrees with a recursive call to the generator we're currently deriving. Finally, `x` remains unconstrained so we instantiate it arbitrarily, like in the `nonempty` tree case.

```
Fixpoint gen_complete (in1 : nat) : G (option Tree) :=
  match in1 with
    | O => ret (Some Leaf)
    | S n => doM! l <- gen_complete n;
             doM! r <- gen_complete n;
             do! x <- arbitrary;
             ret (Some (Node x l r))
  end.
```

The `complete` inductive predicate is particularly well-behaved. First of all, for every possible input depth `m` there exists some tree `t` that satisfies `complete m t`. That will not necessarily hold in the general case. Consider for example an inductive definition that consists of only the `CompleteLeaf` constructor:

```
Inductive half_complete : nat -> Tree -> Prop :=
| CompleteLeaf' : half_complete 0 Leaf.
```

Once again, we will need to pattern match on `m`, and, if `m` is zero, we can proceed as in the previous definition of `complete` to return a `Leaf`. However, if `m` is nonzero there is nothing we can possibly do to return a valid tree that satisfies `half_complete`. This is the reason why our generators return `options` of the underlying type:

```
Definition gen_half_complete (in1 : nat) : G (option Tree) :=
  match in1 with
    | O => ret (Some Leaf)
    | _ => ret None
  end.
```

Secondly, the usage of the input parameter serves as a structurally decreasing parameter for our fixpoint. In the general case that is not necessarily true and we will need to introduce a `size` parameter (like we did in the previous section for simple inductive types), as we will see in the next example.

Binary Search Trees. For a more complex example, consider binary search trees: for every node, each label in its left subtree is smaller than the node label while each label in the right subtree is larger. In Coq code, we could characterize binary search trees whose elements are between two extremal values `lo` and `hi` with the following code:

```
Inductive bst : nat -> nat -> Tree -> Prop :=
| BstLeaf : forall lo hi,
    bst lo hi Leaf
| BstNode : 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 such a search tree since it contains no elements; a Node is such a search tree if its label x is between lo and hi and its left and right subtrees are appropriate search trees as well.

The derived generator (tweaked a bit for readability) is as follows; we explain it below:

```
Definition gen_bst in1 in2 : nat -> G (option Tree) :=
  let fix aux_arb size (in1 in2 : nat) : G (option (Tree)) :=
     match size with
     | O => ret (Some Leaf)
     | S size' =>
       backtrack [(1, ret (Some Leaf))
                 ;(1, doM! x <- arbitraryST (fun x => in1 < x);
                      if (x < in2)? then
                        doM! l <- aux_arb size' in1 x;
                        doM! r <- aux_arb size' x in2;
                        ret (Some (Node x l r))
                      else ret None)]
     end in
 fun size => aux_arb size in1 in2.
```

This generator is bounded: just like in the previous section, we use a natural number size to serve as a limit in the depth of the derivation tree. When size is 0 we are only allowed to use constructors that do not contain recursive calls to the inductive type we're generating. In the binary search tree example, that means that we can only choose the BstLeaf constructor. In that case, we introduce unknowns in1 and in2 that correspond to the inputs to the generation, t that corresponds to the generated tree, as well as two unknowns lo and hi corresponding to the universally quantified variables of the BstLeaf case. We then try to unify in1 with lo, in2 with hi, and t with Leaf. Since lo, hi and t are unconstrained, the unification succeeds and our derived generator returns Some Leaf.

When size is not zero, we have a choice. We can once again choose to satisfy the BstLeaf constructor, which results in the generator returning Some Leaf. We can also choose to try to satisfy the recursive BstNode constructor. After introducing unknowns and performing the necessary unifications, we know that the end product of this sub-generator will be Some (Node x l r). We then proceed to process the constraints that are enforced by the constructor.

To begin with, we encounter lo < x. Since lo is mapped to the input in1, we need to generate x such that x is (strictly) greater than in1. We do that by invoking the typeclass method arbitrarySTfor generating arbitrary natural numbers satisfying the corresponding predicate. Now, when we encounter the x < hi constraint both x and hi are instantiated so we need to *check* whether or not the constraint holds. The notation p? looks for a Dec instance of p to serve as the boolean condition for the if statement. If it does, we proceed to satisfy the rest of the constraints by recursively calling our generator. If not, we can no longer produce a valid binary search tree so we must fail, returning None.

One additional detail in the generator is the use of the backtrack combinator instead of frequency to choose between different constructor options. The backtrack combinator operates exactly like frequency to make the first choice—choosing a generator with type G (option A) based on the induced discrete distribution. However, should the chosen generator fail, it backtracks and chooses a different generator until it either exhausts all options or the backtracking limit.

Nonlinearity. As a last example, we will use an artificial characterization of "good" trees to showcase one last difficulty that arises in the context of dependent inductive types: *non-linear patterns.*

```
Inductive goodTree : nat -> nat -> Tree -> Prop :=
| GoodLeaf : forall n, goodTree n n Leaf.
```

In this example, `goodTree in1 in2 t` only holds if the tree `t` is a `Leaf` and `in1` and `in2` are equal, as shown by the non-linear occurrence of `n` in the conclusion of `GoodLeaf`. If we assume that both `in1` and `in2` will be inputs to our generator, then this will translate to an equality check in the actual generator.

```
Fixpoint gen_good (in1 in2 : nat) size : G (option Tree) :=
  match size with
  | 0 => backtrack [(1, if in1 = in2 ? then ret (Some Leaf)
                        else ret None)]
  | S _ => backtrack [(1, if in1 = in2 ? then ret (Some Leaf)
                          else ret None)]
  end.
```

We can see the equality check `in1 = in2 ?` in the derived generator above. We can also see that the structure of the generator is similar to the one for binary search trees, even though it seems unnecessary. In particular, we encounter calls to `backtrack` with a single element list (which is equivalent to just the inner generator), as well as an unnecessary match on the `size` parameter with duplicated branch code. This uniform treatment of generators facilitates the proof term generation of Section 5. In addition, we could obtain the simpler and slightly more efficient generators by a straight-forward optimization pass.

4 GENERATING GOOD GENERATORS

We now describe the generalized narrowing algorithm more formally.

4.1: Input. Our generation procedure targets simply-typed inductive data that satisfy dependently-typed inductive relations of a particular form. More precisely, we take as input an inductively defined relation R with p arguments of types $A_1, A_2, \cdots, A_p$, where each A_i is a simple inductive type. Each constructor C in the definition of R takes as arguments some number of universally quantified variables ($\overline{x}$) and some preconditions—each consisting of an inductive predicate S applied to constructor expressions (only consisting of constructors and variables) $\overline{e}$; its conclusion is R itself applied to constructor expressions $e_1, e_2, \cdots, e_p$.

$$\begin{array}{c}\texttt{Inductive}\ R : A_1 \to A_2 \to \cdots \to A_p \to \texttt{Prop} \quad := \\ \ldots \quad | \quad C : \forall\, \overline{x},\ \overline{S\ \overline{e}} \to R\ e_1\ e_2\ \cdots\ e_p \quad | \quad \ldots\end{array}$$

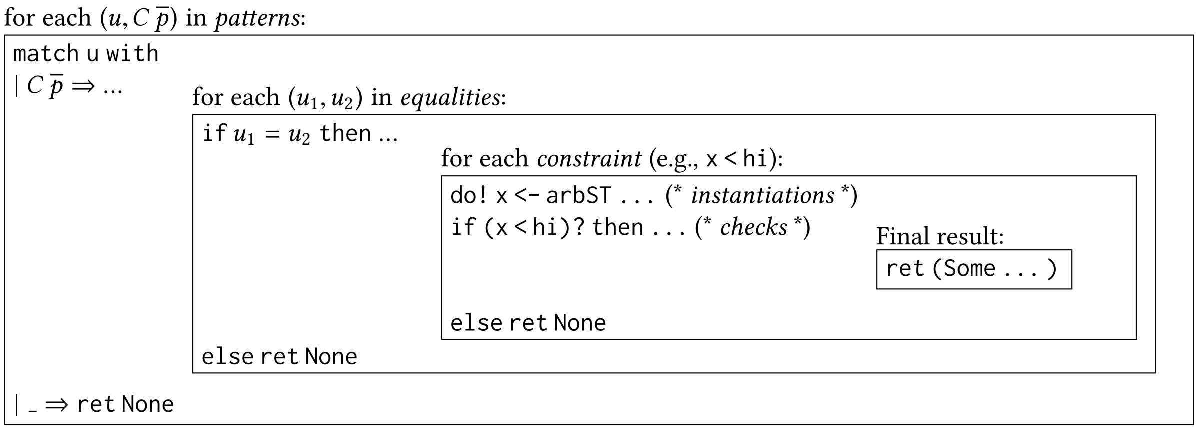
We demonstrate the applicability of this class in practical situations in Section 7 and discuss possible extensions to this format as future work (Section 9).

4.2: Unknowns and Ranges. We first need to formalize *unknowns*, which are used to keep track of sets of potential values that variables can take during generation, similar to logic variables in functional logic programming. One important difference is that sometimes unknowns will be provided as *inputs* to the generation algorithm; this means that they can only take a single fixed value, but that value is not known at derivation time. Looking back at the `complete` trees example, we knew that `in1` would be an input to `gen_complete`. However, when deriving the generator we could not make any assumptions about `in1`: we could not freely unify it with `O` for instance—we had to pattern match against it.

We represent sets of potential values as *ranges*.

$$r := \mathit{undef}_\tau \mid \mathit{fixed} \mid u \mid C\ \overline{r}$$

The first option for the range of an unknown is *undefined* (parameterized by a type). The unknowns we want to generate (such as `tree`, in the binary search tree example) start out with undefined ranges. On the other hand, a range can also be *fixed*, signifying that the corresponding unknown's value serves as an input at runtime (`in1` and `in2` in the binary search tree example). Next, a range of an unknown can also be a different unknown, to facilitate sharing. Finally, a range can be a constructor C fully applied to a list of ranges.



for each $(u, C\ \overline{p})$ in *patterns*:

```
match u with
| C p̄ ⇒ ...
    for each (u1, u2) in equalities:
        if u1 = u2 then ...
            for each constraint (e.g., x < hi):
                do! x <- arbST ... (* instantiations *)
                if (x < hi)? then ... (* checks *)
                    Final result:
                        ret (Some ... )
                else ret None
        else ret None
| _ ⇒ ret None
```

Fig. 1. General Structure of each sub-generator

We use a map from unknowns to ranges, written κ, to track knowledge about unknowns during generation. For each constructor C, we initialize this map with the unknowns that we want to generate mapped to $undef_\tau$ appropriate types τ, the rest of the parameters to R mapped to *fixed*, and the universally quantiified variables of C also mapped to appropriate undefined ranges. For instance, to generate a `tree` such that `bst in1 in2 tree` holds for all `in1` and `in2`, the initial map for the `BstNode` constructor would contain `in1` and `in2` mapped to *fixed*, `tree` mapped to $undef_{\texttt{Tree}}$, and the unknowns `lo`, `hi`, `x`, `l` and `r` introduced by `BstNode` mapped to corresponding undefined ranges:

$$\begin{aligned}\kappa \quad := \quad & (\texttt{in1} \mapsto \textit{fixed}) \oplus (\texttt{in2} \mapsto \textit{fixed}) \\ \oplus \quad & (\texttt{tree} \mapsto \textit{undef}_{\texttt{Tree}}) \\ \oplus \quad & (\texttt{lo} \mapsto \textit{undef}_{\texttt{nat}}) \oplus (\texttt{hi} \mapsto \textit{undef}_{\texttt{nat}}) \oplus (\texttt{x} \mapsto \textit{undef}_{\texttt{Nat}}) \oplus (\texttt{l} \mapsto \textit{undef}_{\texttt{Tree}}) \oplus (\texttt{r} \mapsto \textit{undef}_{\texttt{Tree}})\end{aligned}$$

4.3: Overview. We have already hinted at the general structure of the generation algorithm in Section 3. Let's assume `in1 ... inn` will be the inputs to the generator and that `out1 ... outm` will be the outputs. We then produce a bounded generator that takes `in1` through `inn` as inputs, as well as an additional natural number parameter `size`:

```
Fixpoint aux_arb size in1 ... inn :=
  match size with
  | O => backtrack [ ... (w_C, g_C) ...]
  | S size' => backtrack [ ... (w_C, g_C) ...]
  end.
```

Both when `size` is zero and when it is not, we use `backtrack` to choose between a number of generators. In the latter case, we have one sub-generator g_C for each constructor C. The former case is nearly the same, except that the sub-generators that perform recursive calls to `aux_arb` are filtered out of the list. The weights to backtrack (w_c) can be chosen by the user via lightweight annotations, similar to the local distribution control of Luck (Lampropoulos et al. 2017), as we will see in the evaluation section (7.2). The general structure of each g_C appears in Figure 1.

The outer component of every sub-generator will be a sequence of pattern matches: unification will sometimes signify that we need to match an unknown against a pattern. For instance, in the case of `complete` trees we needed to match `in1` against `O`. Each such pattern match has two branches: one that is considered successful and allows generation to continue; and one that catches all other possible cases and fails (returns `None`).

$$\text{type } \mathit{Pattern} = \mathit{Unknown} \mid C\ \overline{\mathit{Pattern}}$$

$$\text{record } \mathit{UnifyState} = \left\{ \begin{array}{lcl} \mathit{constraints} & : & \mathit{Map\ Unknown\ Range} \\ \mathit{equalities} & : & \mathit{Set}\ (\mathit{Unknown}, \mathit{Unknown}) \\ \mathit{patterns} & : & \mathit{List}\ (\mathit{Unknown}, \mathit{Pattern}) \\ \mathit{unknowns} & : & \mathit{Set\ Unknown} \end{array} \right\}$$

$$\text{type } \mathit{Unify}\ a = \mathit{UnifyState} \rightarrow \mathit{option}\ (a, \mathit{UnifyState})$$

```
$update\ : Unknown \rightarrow Range \rightarrow Unify\ ()$
$update\ u\ r = \lambda s.$
  let $\kappa = constraints(s)$ in
  $Some\ ((),\ s\{constraints \leftarrow \kappa[u \mapsto r]\})$

$equality\ : Unknown \rightarrow Unknown \rightarrow Unify\ ()$
$equality\ u_1\ u_2 = \lambda s.$
  let $eqs = equalities(s)$ in
  $Some\ ((),\ s\{equalities \leftarrow \{u_1 = u_2\} \bigcup eqs\})$

$pattern\ : Unknown \rightarrow Pattern \rightarrow Unify\ ()$
$pattern\ u\ p = \lambda s.$
  let $ps = patterns(s)$ in
  $Some\ ((),\ s\{patterns \leftarrow (u,p) :: ps\})$

$fresh\ : Unify\ Unknown$
$fresh = \lambda s.$
  let $us = unknowns(s)$ in
  let $u = fresh\_unknown\ (us)$ in
  $Some\ (u,\ s\{unknowns \leftarrow u \bigcup us\})$
```

Fig. 2. Unification Monad

After nesting all possible matches, we need to ensure that any equalities raised by the unification hold. In the successful branch of the innermost match (if any), we start a sequence of if-equality-statements. For example, in the case of good trees that were demonstrating non-linear patterns, we checked that `in1 = in2` before continuing with the generation.

The equalities are followed by a sequence of instantiations and checks that are enforced by the hypotheses of C. Looking back at the binary search tree example, we needed to generate a random `x` such that `x` was greater than the lower bound `in1`; we also needed to check whether that generated `x` was less than the upper bound `in2`.

Finally, we combine all the unknowns that we wanted to generate for in a `Some` to return them as the final result. Note that, just like in the `nonEmpty` trees example, we might need to perform a few more instantiations if some unknowns necessary remain completely unconstrained.

4.4: Unification. The most important component of the derivation algorithm is the unification. For every constructor C with conclusion $R\ e_1\ e_2\ \cdots\ e_p$, we convert each e_i to a range and unify it with the corresponding unknown argument of R. For instance, in the binary search tree example, we would unify the `in1`, `in2`, and `tree` unknowns with `lo`, `hi`, and `Node x l r` respectively.

The entire unification algorithm is written inside a state-option monad, presented in Figure 2. To keep track of information about unknowns we use a *Map* from *Unknowns* to *Ranges*; to track necessary equalities—like in the good tree of the previous section—we keep a *Set* of pairs of unknowns; to produce the necessary pattern matches—like in `complete` trees—we gather them in a *List*; finally, to be able to produce fresh unknowns on demand, we keep all existing unknowns in a *Set*.

Each of the four components of the state can be modified through specific monadic actions. The *update* action sets the range of an unknown; the *equality* action registers a new equality check; *pattern* adds a pattern match;

$$
\begin{array}{lcl}
\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}) \\
\\
\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}) \\
\\
\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} \\
\\
\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{array}
$$

$$
\begin{array}{lcl}
\mathit{match}\ u\ (C\ r_1\ \cdots\ r_n) & = & \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{pattern}\ u\ (C\ \overline{p}) \\
\\
\mathit{matchAux}\ (C\ \overline{r}) & = & \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{return}\ (C\ \overline{p}) \\
\mathit{matchAux}\ u & = & r \leftarrow \kappa[u];\ \mathit{case}\ r\ \mathit{of}\ \begin{array}[t]{ll} \ \ \mathit{undef}_\tau & \Rightarrow \mathit{update}\ u\ \mathit{fixed} \\ \mid \mathit{fixed} & \Rightarrow u' \leftarrow \mathit{fresh};\ \mathit{equality}\ u'\ u;\ \mathit{update}\ u'\ u;\ \mathit{return}\ u' \\ \mid u' & \Rightarrow \mathit{matchAux}\ u' \\ \mid C\ \overline{r} & \Rightarrow \overline{p} \leftarrow \mathit{mapM}\ \mathit{matchAux}\ \overline{r};\ \mathit{return}\ (C\ \overline{p}) \end{array}
\end{array}
$$

Fig. 3. Unification algorithm

and *fresh* generates and returns a new unknown. We write $\kappa[u]$ for the action that looks up an unknown, and we write ; for the monadic bind operation and $\bot$ to signify failure (the constant action $\lambda s.\ \mathit{Nothing}$).

The main unification procedure, *unify*, is shown in Figure 3. At the top level, we only need to consider three cases for unification—unknown-unknown, constructor-constructor, and unknown-constructor—because the e_1 through e_p are constructor expressions containing only constructors and variables, which are translated to constructor ranges and unknowns respectively. Most cases are unsurprising; the main important difference from regular unification is the need to handle potentially fixed—but not statically known—inputs.

Case $u_i \mapsto$ undef: If the range of either of the unknowns, say u_1, is undefined, we update κ so that u_1 points to u_2 instead. From that point on, they correspond to exactly the same set of potential values. Consider the `goodTree` example of the previous section, where in the initial map for `GoodLeaf` we have unknowns `in1` and `in2` as inputs to the generator, `tree` as the unknown being generated, and `n` introduced by `GoodLeaf`:

$$\kappa := (\texttt{in1} \mapsto \textit{fixed}) \oplus (\texttt{in2} \mapsto \textit{fixed}) \oplus (\texttt{tree} \mapsto \textit{undef}_{\texttt{Tree}}) \oplus (\texttt{n} \mapsto \textit{undef}_{\texttt{nat}})$$

We first unify `in1` with `n`; since $\texttt{n} \mapsto \textit{undef}_{\texttt{nat}}$ in the initial map, the unification updates that map such that $\texttt{n} \mapsto \texttt{in1}$.

Case $u_i \mapsto u_i'$: If either unknown maps to another unknown we recursively try to unify using the new unknown as input. For example, when we try to unify `in2` with `n` in the updated map for `GoodLeaf`, we recurse and attempt to unify `in1` with `in2`.

Case $u_1 \mapsto C_1\ r_{11}\ \cdots\ r_{1n}$ and $u_2 \mapsto C_2\ r_{21}\ \cdots\ r_{2m}$: If both ranges have some constructor at their head, there are two possibilities: either $C_1 \neq C_2$, in which case the unification fails, or $C_1 = C_2$ and $n = m$, in which case we recursively unify r_{1i} with r_{2i} for all i. We maintain the invariant that all the ranges that appear as arguments to any constructor contain only constructors and unknowns, which allows us to call *unify* and reduce the total number of cases.

The last two cases, dealing with *fixed* ranges, are the most interesting ones.

Case $u_1 \mapsto$ fixed and $u_2 \mapsto$ fixed: If both u_1 and u_2 map to a *fixed* range in κ, then we need to assert that whatever the values of u_1 and u_2 are, they are equal. This will translate to an equality check between u_1 and u_2 in the derived generator. We record this necessary check using *equality* and proceed assuming that the check succeeds, setting one unknown's range to the other. Continuing with the `goodTree` example, when we attempt to unify `in1` and `in2`, both have *fixed* ranges. This results in the equality check $n_1 = n_2$ that appears in `gen_good`.

Case $u_i \mapsto$ fixed and $u_j \mapsto C\ r_1\ \cdots\ r_n$: The last possible configuration pairs a *fixed* range against a constructor range $C\ r_1\ \cdots\ r_n$. This will result in a pattern match in the derived generator. We saw such an example in the previous section in the form of `complete'`. One branch of the match will be against a representation of the range $C\ r_1\ \cdots\ r_n$ and lead to success, while the other branch will terminate the generation with failure in all other cases. To match against $C\ r_1\ \cdots\ r_n$, we will need to convert all of the ranges $\overline{r}$ to patterns $\overline{p}$, while dealing with potentially non-linear appearances of unknowns inside the constructor range. This is done by traversing the ranges $\overline{r}$, applying a helper function *matchAux* to each, and logging the result in the state monad using *pattern*.

If r is itself a constructor C, we need to recursively traverse its ranges, convert them to patterns $\overline{p}$ and combine them into a single pattern $C\ \overline{p}$. If r is an unknown u, we look up its range inside the current map. If it is undefined we can use u as the bound variable in the pattern; we update the binding of u in the map to be *fixed*, as it will be extracting information out of the fixed discriminee. On the other hand, if the range is *fixed*, we need to create a *fresh* unknown u', use that as the pattern variable and then enforce an equality check between u and u'. Finally, the unknown and constructor cases result in appropriate recursions.

4.5: Handling hypotheses. Another important part of the derivation of a generator for a single constructor C is handling all of its hypotheses. Given a hypothesis of the form $S\ e_1\ e_2\ \cdots\ e_m$, we once again identify a few different cases.

If there is exactly one undefined variable amongst the e_i, we need to instantiate it. That translates either to a call to the generic `arbitraryST` function, or to a recursive call to the currently derived generator. The bst predicate provides examples of both: after the unification is complete, the map κ will have the following form:

$$\begin{array}{rcl} \kappa & := & (\texttt{in1} \mapsto \textit{fixed}) \oplus (\texttt{in2} \mapsto \textit{fixed}) \oplus (\texttt{tree} \mapsto \texttt{Node x l r}) \\ & \oplus & (\texttt{lo} \mapsto \texttt{in1}) \oplus (\texttt{hi} \mapsto \texttt{in2}) \oplus (\texttt{x} \mapsto \textit{undef}_{\texttt{Nat}}) \oplus (\texttt{l} \mapsto \textit{undef}_{\texttt{Tree}}) \oplus (\texttt{r} \mapsto \textit{undef}_{\texttt{Tree}}) \end{array}$$

When processing the hypothesis `lo < x`, the unknown `lo` maps to `in1`which in turn is *fixed*, while `x` is still undefined. Thus, to generate `x` such that `lo<x` holds, we need to invoke the `arbitraryST` method of `GenSuchThat` for `(fun x => in1 < x)`. After processing this constraint, the range of `x` becomes to *fixed*: we know that it has a concrete value but not what it is. For all intents and purposes it can be treated as if it was an input to the generation from this point on.

$$\begin{array}{rcl} \kappa & := & (\texttt{in1} \mapsto \mathit{fixed}) \oplus (\texttt{in2} \mapsto \mathit{fixed}) \oplus (\texttt{tree} \mapsto \texttt{Node x l r}) \\ & \oplus & (\texttt{lo} \mapsto \texttt{in1}) \oplus (\texttt{hi} \mapsto \texttt{in2}) \oplus (\texttt{x} \mapsto \mathit{fixed}) \oplus (\texttt{l} \mapsto \mathit{undef}_{\texttt{Tree}}) \oplus (\texttt{r} \mapsto \mathit{undef}_{\texttt{Tree}}) \end{array}$$

Therefore, when processing the `bst lo x l`, only `l` is unconstrained. However, since generating `l` such that `bst lo x l` holds is exactly the generation mode we are currently deriving, we just make a recursive call to `aux_arb` to generate `l`.

The second possibility for a hypothesis is that all expressions e_i are completely fixed, in which case we can only check whether this hypothesis holds. For example, when we encounter the `x < hi` constraint, both `x` and `hi` have already been instantiated and therefore we need to check whether `x < hi` holds at runtime, using the `dec` method of the decidability typeclass.

A final possibility is that a hypothesis could contain multiple undefined unknowns. Deciding which of them to instantiate first and how many at a time is a matter of heuristics. For example, if in the constraint `bst lo hi t`, if all of `lo`, `hi` and `t` were undefined, we could pick to make a call to `arbitraryST bst`, or we could instantiate arguments one at a time. In our implementation, we prioritize recursive calls whenever possible; we leave further exploration and comparison of different heuristics as future work.

4.6: Assembling the Final Result. After processing all hypotheses we have an updated constraint map κ, where, compared to the constraint map after the unification, some unknowns have had their ranges *fixed* as a result of instantiation. However, there might still be remaining unknowns that are undefined. Such was the case for the `nonEmpty` tree example where `x`, `l` and `r` were all still undefined. Thus, we must iterate through κ, instantiating any unknowns u for which $\kappa[u] = \mathit{undef}$. To complete the generator g_C for a particular constructor, we look up the range of all unknowns that are being generated, convert them to a Coq expression, group them in a tuple and return them.

4.7: Putting it All Together. A formal presentation of the derivation for a single constructor is shown in Figure 4. Here, for simplicity of exposition, we allow only a single output *out*. In general, even though our implementation of the algorithm deals with a single output as well, the algorithm presented in this section can handle an arbitrary number of outputs.

Given an inductive relation R and a particular constructor $C : \forall\, \overline{x},\ \overline{S\ \overline{e}} \rightarrow P\ e_1\ e_2\ \cdots\ e_p$, our goal is to generate *out* such that for all $\overline{in}$, the predicate $R\ e'_1\ e'_2\ \ldots\ e'_p$ holds via constructor C, where the e's are constructor expressions containing only variables in $\{out\} \bigcup \overline{in}$. First, we create an initial map κ as described in Paragraph 4.2. We use it to construct an initial state *st* for the unification monad (Paragraph 4.4), where the *patterns* and *equalities* fields are empty, while the *unknowns* field holds $\overline{in}$, *out* and all universally quantified variables of C. We then evaluate a sequence of monadic actions, each one attempting to unify e_i with its corresponding e'_i. If at any point the unification fails, the constructor C is not inhabitable and we fail. If it succeeds, we proceed to produce all of the nested pattern matches and equalities in order (*emit_patterns* and *emit_equalities*), as described Paragraph 4.3. Afterwards, we process all the hypotheses using *emit_hypotheses* as described in Paragraph 4.5, emitting instantiations or checks as appropriate, while updating the constraint set κ. Finally, we complete the generation by instantiating all unknowns that are still undefined and constructing the result by reading off the range of *out* in the final constraint set 4.7.

5 GENERATING CORRECTNESS PROOFS

This section describes how we automatically generate proofs that our generators are sound and complete with respect to the inductive predicates they were derived from. Following the translation validation approach of Pnueli et al. (1998), rather than proving once and for all that *every* generator we build is guaranteed to be correct, we build

```
let $\kappa = \overline{in \mapsto fixed} \oplus out \mapsto undef_{\tau_{out}} \oplus \overline{x \mapsto undef_{\tau_x}}$ in
let $st = \left\{ \begin{array}{lcl} constraints & = & \kappa \\ equalities & = & \emptyset \\ patterns & = & [] \\ unknowns & = & \overline{in} \bigcup \{out\} \bigcup \overline{x} \end{array} \right\}$ in
case $runUnifyState\ \{unify\ e_1\ e'_1;\ unify\ e_2\ e'_2;\ \ldots;\ unify\ e_p\ e'_p\}\ st$ of
  | $None \rightarrow$ $\boxed{\texttt{ret None}}$
  | $Some\ (st', ()) \rightarrow emit\_patterns\ (patterns\ st')\ (equalities\ st')\ (constraints\ st')$
```

```
$emit\_patterns\ []\ eqs\ \kappa = emit\_equalities\ eqs\ \kappa$
$emit\_patterns\ ((u, p) :: ps)\ eqs\ \kappa =$
    ┌─────────────────────────────────────┐
    match $u$ with
    | $p$ => $emit\_patterns\ ps\ eqs\ \kappa$
    └─────────────────────────────────────┘
```

```
$emit\_equalities\ []\ \kappa = emit\_hypotheses\ (\overline{S\ \overline{e}})\ \kappa$
$emit\_equalities\ ((u_1, u_2) :: eqs)\ \kappa =$
    ┌─────────────────────────────────────┐
    if $(u_1 = u_2)$? then $emit\_equalities\ eqs\ \kappa$
    else ret None
    └─────────────────────────────────────┘
```

```
$emit\_final\_call\ u_k =$
    if $u_k \sim out$ then $\boxed{\texttt{doM! } u_k \texttt{ <- aux\_arb size'}}$
    else $\boxed{\texttt{doM! } u_k \texttt{ <- arbitraryST } (\lambda u_k.\ S\ \overline{e})}$
```

```
$emit\_hypotheses\ []\ \kappa = final\_assembly\ \kappa$
$emit\_hypotheses\ ((S\ \overline{e}) :: ss)\ \kappa =$
  if $\forall\ u \in \overline{e}.\ \kappa[u] \neq undef$ then
    ┌─────────────────────────────────────┐
    if $(S\ \overline{e})$? then $emit\_hypotheses\ ss\ \kappa$
    else ret None
    └─────────────────────────────────────┘
  else let $\{u_1, \ldots, u_k\} = \{u \in \overline{e} \mid \kappa[u] = undef\}$ in
    ┌─────────────────────────────────────┐
    do! $u_1$ <- arbitrary;
    ...
    do! $u_{k-1}$ <- arbitrary;
    $emit\_final\_call\ u_k$
    $emit\_hypotheses\ ss\ \kappa[u_i \leftarrow fixed]_{i=1}^{k}$
    └─────────────────────────────────────┘
```

```
$final\_assembly\ \kappa =$
  let $\{u_1, \ldots, u_f\} = \{u \mid \kappa[u] = undef\}$ in
  let $\kappa' = \kappa[u_i \leftarrow fixed]_{i=1}^{k}$ in
    ┌─────────────────────────────────────┐
    do! u <- arbitrary;
    ret (Some $emit\_result\ \kappa'\ out\ \kappa'[out]$)
    └─────────────────────────────────────┘
```

```
$emit\_result\ \kappa\ u\ u' = emit\_result\ \kappa\ u'\ \kappa[u']$
$emit\_result\ \kappa\ u\ fixed = \boxed{u}$
$emit\_result\ \kappa\ u\ (C\ r_1\ \ldots\ r_k) =$
    $\boxed{C\ (emit\_result\ \kappa\ \_\ r_1)\ \ldots\ (emit\_result\ \kappa\ \_\ r_k)}$
```

Fig. 4. Derivation of one case of a generator g_C (for a single constructor C), in pseudo-code. Boxes delimit "quasi-quoted" Coq generator code to be emitted. Inside boxes, *italic* text indicates "anti-quoted" pseudo-code whose result is to be substituted in its place.

a proof term certifying that each specific generator is correct at the same time as we build the generator itself. In fact, the same algorithm that is used to compile generators from inductive predicates is also used to compile their corresponding proofs of correctness. We leverage an existing verification framework for QuickChick, designed to allow users to (manually) prove soundness and completeness of generators built from QuickChick's low-level primitives (Paraskevopoulou et al. 2015).

This verification framework assigns semantics to each generator by mapping it to its set of outcomes, i.e. the elements that have non-zero probability of being generated. This enables proving that all the elements in a set of outcomes satisfy some desired predicate (soundness), and that all the elements that satisfy the predicate are in the set of outcomes (completeness). To ease reasoning about user-defined generators, QuickChick provides a library of lemmas that specify the behavior of built-in combinators.

We leverage this framework to specify the set of outcomes of derived generators: given an inductive relation and some input parameters, it should be exactly the set of elements that satisfy the inductive relation. Automatic generation of proofs is analogous to generation of generators and is done using the same algorithm. Just as generators are derived by composing generator combinators that we select by examining the structure of the inductive predicate, proofs are derived by composing the corresponding correctness lemmas that are provided by QuickChick. We glue these proof components together in order to obtain proofs for unsized generators using typeclass resolution, just as we did to obtained unsized generators. To enable this, we extend the typeclass infrastructure of QuickChick to encode properties of generators as typeclasses and we automatically generate instances of these classes for the derived generators.

Subsection 5.1 briefly describes QuickChick's verification framework, focusing on the proof generation machinery. 5.2 outlines the structure of the generated proofs and describes all the terms, definitions, and proofs that we need to generate in order to obtain the top-level correctness proof. 5.3 describes the extensions to the typeclass infrastructure of QuickChick that we made in order to facilitate proof generation.

5.1 Verification Framework

QuickChick assigns semantics to generators by mapping them to the set of values that have non-zero probability of being generated. Recall from section 2.1 that generators are functions mapping a random seed and a natural number to an element of the underlying type. The semantics of a generator for a given size parameter is exactly the values that can be generated for this particular size parameter.

$$[\![g]\!]_s = \{\ x \mid \exists r,\ g\ s\ r = x\ \}$$

We can then define the semantics of a generator by taking the union of these sets over all possible size parameters.

$$[\![g]\!] = \bigcup_{s \in \mathbb{N}} [\![g]\!]_s$$

It may seem as though we could have skipped the first definition and inlined its right-hand side in the second. However, by separating out the first definition we can additionally characterize the behavior of generators with respect to the size parameter. For instance, we can define the class of size-monotonic generators, whose set of outcomes for a given size parameter is included to the set of outcomes for every larger size parameter.

$$\texttt{sizeMonotonic}\, g \overset{\text{def}}{=} \forall s_1\ s_2,\ s_1 \leq s_2 \rightarrow [\![g]\!]_{s_1} \subseteq [\![g]\!]_{s_2}$$

Another useful class of generators is bound-monotonic generators, i.e., bounded generators that behave monotonically with respect to their bound parameter. Recall that bounded generators are parameterized by a natural number which bounds the size of the generated terms.

$$\texttt{boundMonotonic}\, g \overset{\text{def}}{=} \forall s\ s_1\ s_2,\ s_1 \leq s_2 \rightarrow [\![g\ s_1]\!]_s \subseteq [\![g\ s_2]\!]_s$$

Together, these characterizations allow us to obtain convenient specifications for combinators. To support reasoning about size-monotonicity properties, QuickChick encodes them as typeclasses and provides lemmas (encoded as typeclass instances) that various generator combinators are size monotonic if all the involved generators are size monotonic. For instance, here is the lemma for monadic binding:

$$\frac{\texttt{sizeMonotonic}\, g \qquad \forall x \in [\![g]\!],\ \texttt{sizeMonotonic}\ (f\ x)}{\texttt{sizeMonotonic}\ (g >>= f)}\ \textsc{monBind}$$

There is a similar lemma that guarantees that `sized` is size monotonic but it requires both bound and size monotonicity for the bounded generator.

$$\frac{\texttt{sizeMonotonic}\ g \qquad \forall s,\ \texttt{boundMonotonic}\ (g\ s)}{\texttt{sizeMonotonic}\ (\texttt{sized}\ g)}\ \textsc{monSized}$$

To prove that `sized` is monotonic, we also need a second premise that requires the bounded generator to be bound monotonic. This is because `sized` will use the internal size parameter of `G` to instantiate the bound parameter of the generator.

To support reasoning about generators, QuickChick provides a library of lemmas that specify the semantics of generator combinators and can be used to compositionally verify user defined generators. These lemmas can be seen as a proof theory for the `G` monad; one can apply them in order to build derivations that computations in this monad are correct.

The simplest example of a correctness lemma is the one of `ret`. Unsurprisingly, the semantics of the return of the `G` monad is just a singleton set.

$$\frac{}{[\![\texttt{ret}\ x]\!] = \{\ x\ \}}\ \textsc{semRet}$$

The lemma for monadic bind is more interesting. In particular, the expected specification that composes the set of outcomes of the two generators using an indexed union is true, but under the additional requirement that the generators involved are size monotonic.

$$\frac{\texttt{sizeMonotonic}\ g \qquad \forall x \in [\![g]\!],\ \texttt{sizeMonotonic}\ (f\ x) \qquad [\![g]\!] = s \qquad \forall x \in s,\ [\![f\ x]\!] = h\ x}{[\![g \mathbin{>>=} f]\!] = \bigcup_{x \in s} h\ x}\ \textsc{semBind}$$

The intuition behind this requirement (Paraskevopoulou et al. 2015) is that the set on the left-hand side of semBind contains elements that are generated when the same size parameter is threaded to both generators, whereas the right-hand side indexes over elements that have been generated by g when the size parameter ranges over all natural numbers. To address this mismatch, we use monotonicity. In particular, to obtain the right to left inclusion we can pick a witness for the size parameter that is greater than both of the size parameters we obtain as witnesses from the hypothesis, such as their sum (or max) and then use monotonicity to prove the inclusion.

The correctness lemma for `sized` is crucial for proof generation, as it gives us proofs about unbounded generators. It states that the semantics of the combinator is the union of the sets of outcomes of the bounded generator indexed over all natural numbers.

$$\frac{\texttt{boundMonotonic}\ g \qquad (\forall x \in \mathbb{N},\ \texttt{sizeMonotonic}\ (g\ x)) \qquad \forall x \in \mathbb{N},\ [\![g\ x]\!] = f\ x}{[\![\texttt{sized}\ g]\!] = \bigcup_{x \in \mathbb{N}} f\ x}\ \textsc{semSized}$$

The lemma requires that the bounded generator is size monotonic for all bounds and, in addition, that it is monotonic in the bound parameter itself. These conditions are required because the set on the left-hand size of the specification contains elements that are generated from g using same number for the size and the bound, whereas in the right-hand side the bound and the size parameter range independently over natural numbers. As in the case of the monadic bind, we can work around this mismatch using monotonicity.

5.2 Proof Generation

This section describes the proof terms that we generate for each derived generator. To describe the structure of the constructed proof terms, we will use the generator for binary search trees presented in 3 as a running example. The terms themselves have the same structure as the generators and are generated using using the same algorithms, replacing the generator building blocks with their proof counterparts. For brevity, we assume a single output of the generation procedure; we can easily encode multiple outputs using tuples.

Top-level proof. Let $R : A \to Prop$ be an inductive predicate and g : `option` A a derived generator for this predicate. We want to generate proofs that g is sound and complete with respect to this predicate, i.e., that the set of outcomes of the generator is exactly the elements that satisfy P:

$$\texttt{isSome} \cap [\![g]\!] = \texttt{Some}[P]$$

Since our generators can fail (their return type is `option` A), we need to take the image of P under `Some`. We also remove `None` from the set of outcomes of g by intersecting it with `isSome`, i.e., the set of elements whose outermost constructor is `Some`.

In the case of binary search trees, this amounts to saying that the set of outcomes of the unbounded generator (which we obtain using `sized`; it is automatically derived by typeclass resolution) is exactly the set of trees that satisfy the `bst` predicate for some given inputs.

$$\forall\, in_1\ in_2,\ \texttt{isSome} \cap [\![\texttt{sized}\ (\texttt{gen\_bst}\ in_1\ in_2)]\!] = \texttt{Some}[\texttt{bst}\ in_1\ in_2]$$

As expected, to generate proofs about unbounded generators we have to first generate proofs about bounded generators. These proofs can be then lifted using the specification of `sized` that we saw in the previous section.

Proofs for Bounded Generators. Before deriving correctness proofs for bounded generators, we first need to settle on a specification. To this end, for each inductive definition for which we derive a generator, we generate automatically an operator, which we call `iter`, that maps a natural number to the set of elements that inhabit the inductive relation and whose proof has height less or equal to the given natural number. This set will serve as a specification for the bounded generator for a given size parameter.

This operator has exactly the same shape as the generator, and it is obtained using the same algorithm; the only thing that changes is that, instead of using the combinators of the `G` (`option` $-$) monad, we use those of the `set` monad. For instance, in the case of the `bst` predicate the `iter` operator looks like the following:

$$\texttt{iter\_bst}\ in_1\ in_2\ s = \begin{cases} \{\ \texttt{Leaf}\ \} & \text{if } s = 0 \\ \{\ \texttt{Leaf}\ \} \cup \bigcup_{x > in_1} \texttt{if}\ x < in_2\ \texttt{then} & \text{if } s = s' + 1 \\ \qquad \bigcup_{l \in (\texttt{iter\_bst}\ in_1\ x\ s')} \bigcup_{r \in (\texttt{iter\_bst}\ x\ in_2\ s')} \{\ \texttt{Node}\ x\ l\ r\ \} & \\ \qquad \texttt{else}\ \emptyset & \end{cases}$$

The parallels with the generator stand out: we can obtain this by replacing `ret` (`Some` $-$) with the singleton set (i.e., the return of the `set` monad), `bind` with indexed union (i.e., the bind of the `set` monad), and `ret None` with empty set (i.e., the fail action of the `set` monad).

Using `iter` we can accurately characterize the set of outcomes of a bounded generator:

$$\texttt{isSome} \cap [\![g\ n]\!] = \texttt{Some}[\texttt{iter}\ n]$$

The proof term for this proposition also has the same structure as the generator, but this time, instead of monadic combinators, we use the corresponding proof rules. Since we only care to specify the `Some` part of the set of outcomes of the generators, we can use slightly modified proof rules that require a weaker notion of generator monotonicity. In particular, the new rules only require the generator to be monotonic in the `Some` part of its set of outcomes. This is captured by the following two definitions.

$$\texttt{sizeMonotonicOpt}\ g \overset{\text{def}}{=} \forall s_1\ s_2,\ s_1 \leq s_2 \to \texttt{isSome} \cap [\![g]\!]_{s_1} \subseteq \texttt{isSome} \cap [\![g]\!]_{s_2}$$

$$\texttt{boundMonotonicOpt}\ g \overset{\text{def}}{=} \forall s\ s_1\ s_2,\ s_1 \leq s_2 \to \texttt{isSome} \cap [\![g\ s_1]\!]_s \subseteq \texttt{isSome} \cap [\![g\ s_2]\!]_s$$

Using the above definitions we can formulate the alternative proof rules. Below are examples of reformulated lemmas for `bind` and `size`.

$$\frac{\begin{array}{c}\texttt{sizeMonotonicOpt}\ g \\ \forall x,\ \texttt{sizeMonotonicOpt}\ (f\ x) \qquad [\![g]\!] = s \qquad \forall x,\ \texttt{isSome} \cap [\![f\ x]\!] = h\ x\end{array}}{\texttt{isSome} \cap [\![g \mathbin{>>=} f]\!] = \bigcup_{x \in s} h\ x}\ \textsc{semOptBind}$$

$$\frac{\texttt{boundMonotonicOpt}\ g \qquad \forall n \in \mathbb{N},\ \texttt{sizeMonotonicOpt}\ (g\ n)) \qquad \forall n,\ \texttt{isSome} \cap [\![g\ n]\!] = h\ n}{\texttt{isSome} \cap [\![\texttt{sized}\ g]\!] = \bigcup_{n \in \mathbb{N}} h\ n}\ \textsc{semOptSized}$$

Our goal is to lift specification from bounded to unbounded generators using the corresponding lemma for `sized`. To that end, we need a proof that the union of these sets produced by `iter` over all natural numbers is exactly the set of elements that satisfy the predicate.

$$\bigcup_{n \in \mathbb{N}} \texttt{iter}\ n = P$$

The above proof also requires us to generate a proof that these sets operators are monotonic in the size parameter.

$$\forall n_1\ n_2,\ n_1 \leq n_2 \to \texttt{iter}\ n_1 \subseteq \texttt{iter}\ n_2$$

Monotonicity proofs. As described above, in order to produce correctness proofs we need to also produce monotonicity proofs for the unbounded generators. These proofs are used in both constructing the correctness proofs for bounded combinators, as well as lifting them to unbounded ones. In order to be able to use the generators as individual components in other derived generators that come with correctness proofs, we also lift size monotonicity proofs to unbounded generators. As in previous cases, this is done using the corresponding lemma for `sized`. Again, we automate this process by providing the appropriate typeclass instances. Note that the choice to generate proofs of this weaker notion of monotonicity is not essential; we could have generated proofs of full monotonicity instead. However, we opted for this weaker notion as it significantly simplifies the proof of bound monotonicity.

5.3 Typeclasses for Proof Generation

As we did for generators in Section 4, we rely on typeclasses to connect individual proof components and lift specifications to unbounded generators. In this subsection we describe the extensions to the typeclass infrastructure of QuickChick presented in Section 2, which are needed in order to achieve this. We use Coq notation so that we can display the actual typeclass definitions; the notation `:&:` denotes set intersection and the notation `@:` the image of a function over some set.

Monotonicity. First we extend the typeclass hierarchy to encode size and bound monotonicity properties.

```
Class SizeMonotonicOpt {A} (g : G (option A)) :=
  { monotonic_opt :
      forall s1 s2,
        s1 <= s2 -> isSome :&: semGenSize g s1 \subset isSome :&: semGenSize g s2 }.

Class BoundMonotonicOpt {A} (g : nat -> G (option A)) :=
  { sizeMonotonicOpt :
      forall s s1 s2,
        s1 <= s2 ->
        isSome :&: semGenSize (g s1) s \subset isSome :&: semGenSize (g s2) s }.
```

We automatically generate proofs that derived bounded generators are bound and size monotonic by explicitly constructing the proof term and we automatically create instances of these classes. Size monotonicity can then be derived for unbounded generators using the following provided instance.

```
Instance SizeMonotonicOptOfBounded (A : Type) (P : A -> Prop)
         (H1 : GenSizedSuchThat A P)
         (H2 : forall s : nat, SizeMonotonicOpt (arbitrarySizeST P s))
         (H3 : BoundMonotonicOpt (arbitrarySizeST P))
: SizeMonotonicOpt (arbitraryST P).
```

Given a `GenSizedSuchThat` instance for a predicate *P* (`H1` above), which provides access to a constrainted bounded generator `arbitrarySizeST P`, and instances of size and bound monotonicity for this generator (`H2` and `H3`), we can obtain an instance of size monotonicity for unbounded generator for this predicate, `arbitraryST P`, which is also obtained automatically by the corresponding instance.

Set Operators. To express the correctness property of generators we introduce a typeclass that gives a generic interface to predicates which are equipped with an `iter` operator.

```
Class Iter {A : Type} (P : A -> Prop) :=
  { iter : nat -> set A;
    iter_mon : forall n1 n2, n1 <= n2 -> iter n1 \subset iter n2;
    iter_spec : \bigcup_(n : nat) (iter n) ≡ P }.
```

Correctness. We can define a subclass of the above class, which is used to characterize bounded generators that are correct with respect to a predicate.

```
Class BoundedSuchThatCorrect {A : Type} (P : A -> Prop) {Iter A P}
      (g : nat -> G (option A)) :=
  { boundedCorrect : forall s, isSome :&: semGen (g s) ≡ Some @: (iter s) }.
```

In the above, we are requiring that *P* is an instance of the `Iter` class in order to be able to use `iter` to express the correctness property. Following our usual practice, we also define a class for correct unbounded generators.

```
Class SuchThatCorrect {A : Type} (P : A -> Prop) (g : G (option A)) :=
  { correct : isSome :&: semGen g ≡ Some @: P }.
```

As before, we automatically generate instances for correctness of bounded generators by proving the proof terms, and we then lift them to unbounded generators by adding the corresponding instance.

```
Instance SuchThatCorrectOfBounded (A : Type) (P : A -> Prop)
         (H1 : GenSizedSuchThat A P)
         (H2 : Iter P)
         (H3 : forall s : nat, SizeMonotonicOpt (arbitrarySizeST P s))
         (H4 : BoundMonotonicOpt (arbitrarySizeST P))
         (H5 : SizedSuchThatCorrect P (arbitrarySizeST P))
: SuchThatCorrect P (arbitraryST P).
```

The above instance is similar to the one for monotonicity but it additionally requires an instance for correctness of the unbounded generator (`H5`). It also requires an instance of the `Iter` class for `P` (`H2`). This instance is required as an (implicit) argument to the instance of correctness and also in the proof itself as it provides the specification of `iter`.

6 IMPLEMENTATION

Our initial implementation of the generator derivation algorithm interfaced directly with Coq's internals. But, even for the simply-typed inductive generators of Section 2, this was neither an extensible nor a maintainable approach. Coq's term data structure, for example, contains far too much type information that our application does not care about. Similarly, the internal functions that produce Coq expressions take more arguments that we need, in order to accurately populate the rich data structure.

To facilitate deriving generators and proof terms (as well as other needed infrastructure such as `Show` instances and shrinking functions that we haven't touched on in this paper), we wrote a small generic programming framework consisting of two parts: a high level representation of the class of inductive terms we target and a small DSL for producing Coq expressions. We represent the class we target with the following datatype:

```
type dep_type =
  | DArrow   of dep_type * dep_type          (* Unnamed arrows *)
  | DProd    of (var * dep_type) * dep_type   (* Binding arrows *)
  | DTyParam of ty_param                      (* Type parameters *)
  | DTyCtr   of ty_ctr * dep_type list        (* Type Constructor *)
  | DCtr     of constructor * dep_type list   (* Data Constructor *)
  | DTyVar   of var                           (* Use of a type variable *)
```

The representation is relatively standard. Note that arrows and products are treated as a top-level constructors, since they are of particular importance: arrows can be used to represent side-conditions and products to capture the universally quantified variables of each constructor. Each type above (like `var` or `constructor`) is an opaque wrapper around Coq identifiers, completing the separation of the generic library user from Coq internals.

For our term-building DSL, we provide higher-order abstract syntax combinators. To guarantee well-scopedness of generated terms, we handle fresh name generation internally. For example, the combinator for a local recursive fixpoint `let fix ... in` (like the one used to define `aux_arb` in the previous sections) is:

```
val gRecIn : string -> string list -> (var * var list -> coq_expr) ->
             (var -> coq_expr) -> coq_expr
```

To construct a fixpoint, we require a `string`, the base name for the recursive fixpoint, and a list of strings, the base name for each argument. Internally, we convert each such string to a fresh Coq identifier. These identifiers can be used to construct other Coq expressions: inside the fixpoint we can use the opaque representation of both the fixpoint name and its arguments; in the continuation we can only use the fixpoint itself.

7 EVALUATION

We evaluate two aspects of our generators: the applicability of the restricted class of inductive types we target (7.1) and the efficiency of the derived generators compared to handwritten ones (7.2).

7.1 QuickChecking Software Foundations

To evaluate the applicability of our algorithm we tried to automatically test a large body of specifications that are representative of those commonly used in verifying properties of programming languages. Such a body of specifications can be found in Software Foundations (Pierce et al. 2016), a machine-checked textbook for programming language theory and verification using Coq. We attempted to automatically test every theorem or lemma in the suggested main course of the textbook, all the way through the simply typed lambda calculus chapters.

Our findings are summarized in Figure 5. To avoid skewing our findings, we separately count certain classes of examples. Of the 232 nontrivial (non-unit-test) theorems we considered, 194 (84%) are directly amenable

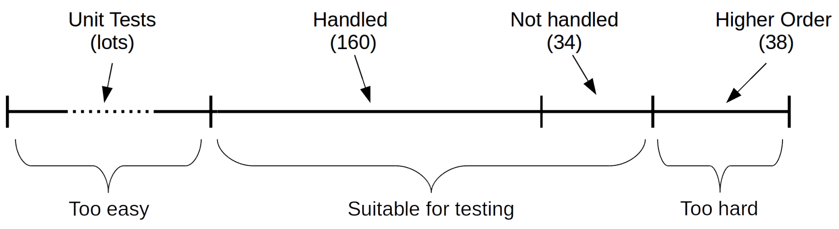


Fig. 5. Evaluation Results

to PBRT; the 38 remaining theorems deal with generation for higher-order properties, which we deem too difficult for automatic test-case generation (we give examples below). Of the 194 theorems we believed “should be testable,” 160 (83%) could be tested using our implemented algorithm. This demonstrates that the class of inductive propositions targeted by our narrowing generalization is broad enough to tackle many practical cases in the *Software Foundations* setting. The rest of this section discusses the different classes of theorems we considered and our methodology for testing each one.

First of all, *Software Foundations* incorporates a large number of unit tests, like the following test for disjunction:

```
Example test_orb1:  (orb true false) = true.
```

Such examples are trivially checkable and unininteresting from a generation perspective.

On the other hand, a class of lemmas that are completely out of scope of generation techniques deal with universally quantified higher-order properties. Consider the following canonical example of a Hoare triple:

```
Theorem hoare_seq : forall P Q R c1 c2,
     {{Q}} c2 {{R}}  ->  {{P}} c1 {{Q}}  ->  {{P}} c1;;c2 {{R}}.
```

Testing such a property would require generating arbitrary elements of type `state` -> `Prop`, which is beyond current automatic random generation techniques. For context, the number of higher-order properties we excluded were 38; 36 of them came from the Logic and Hoare logic chapters that heavily use quantification over `Prop`s.

Finally, a third class of properties that could be interesting from a generation perspective but are a poor fit for property-based random testing are existential properties. For example, consider progress for a type system:

```
Conjecture progress : forall t T,  |- t \in T  ->  value t \/ exists t', t ===> t'.
```

While generating `t` and `T` such that `t` has type `T` is both interesting and possible within the extension of QuickChick presented in this paper, it is not possible to decide whether the conclusion of the property holds! However, most of the time, it is possible to rewrite existential conclusions into decidable ones. For example, for a deterministic step relation, we could write a partial `step` function and rewrite the conclusion to check whether the term `t` can actually take a step: `isSome (step t)`.

With the above in mind we proceeded to automatically derive generators for all simple inductive types, generators for different modes for inductive relations, as well as proofs for both. We completely elided unit tests, counted (but otherwise ignored) properties that required generation of higher order properties, and converted conclusions to decidable when necessary. We then turned each property into a `Conjecture`—an automatically admitted property—and attempted to test it with QuickChick. For example, the preservation property became:

```
Conjecture preservation : forall t t' T,  |- t \in T  ->  t ===> t'  ->   |- t' \in T.
QuickChick preservation.
```

This simulates a common workflow in interactions with the Coq proof assistant: in order to prove a large theorem (e.g. type safety), provers often `Admit` smaller lemmas to construct the proof, discharging them afterwards. However, admitting a lemma that is too strong can lead to a lot of wasted effort and frustration. Using QuickChick, users can uncover bugs early on while building confidence in such conjectures.

For a small portion of the theorems we allowed minor changes (i.e., converting preconditions like `beq_nat n1 n2` to `n1 = n2`). Overall, we only performed one major change: converting Software Foundation `Maps` from a functional representation to an explicit list-based one. A functional representation for maps is convenient for looking up element's associations, but—unless the domain of the function is bounded—makes it completely impossible to do the reverse. That requirement is very common in generation—for instance, picking a variable with a specific type from a given context. Moreover, a lot of properties needed to decide equivalence of two maps, which is also impossible in a functional representation. Therefore, we changed maps to a more generation-friendly variant. The new map code was similar in length with respect to the old one (∼ 40 lines), including automatic derivations of generators and decidability instances, but resulted in many syntactic changes across the rest of the chapters.

7.2 QuickChecking Noninterference

To evaluate the efficiency of our approach, we conducted a case study comparing the runtime performance of our derived generators against carefully tuned handwritten ones: the generators for the information-flow control experiments in Hriţcu et al. (2013, 2016), which generate indistinguishable pairs of machine states using Haskell's QuickCheck to discover noninterference violations. These generators had already been ported to QuickChick, where they formed the main case study for the proof generation framework of Paraskevopoulou et al. (2015). There, they were systematically evaluated using a rigorous mutation testing methodology, which we reused here to ensure that our derived generators had roughly the same bug-finding capabilities. Our experiments showed that the derived generators were 1.75× slower than the corresponding handwritten ones, while producing the same distribution and bugfinding performance.

In more detail: Dynamic information-flow controltags data values with security levels, called *labels*, and uses them to prevent flows from *high* (secret) to *low* (public) data. In particular, Hriţcu et al. (2013) enhanced a simple, low-level stack machine with label information and tested it for *termination-insensitive noninterference*: given two *indistinguishable* machine states, i.e. states that differ only in high data, running them to completion should yield indistinguishable states. This is a prototypical example of a conditional property: if we were to generate pairs of arbitrary machine states and discard those that are not indistinguishable, we would almost never exercise the conclusion! Instead, Hriţcu et al. (2013) generated a single arbitrary machine state first and then *varied* that state to produce a new one that was indistinguishable (by construction).

For our evaluation, we focused on a stronger property (also considered by Hriţcu et al. (2013)), *single-step noninterference*, which only runs both machines for a single step. As Hriţcu et al. (2013) showed, this makes generators for valid initial states substantially simpler: since only one instruction will be executed, memories do not need to be longer than two elements (no more than one element can be accessed by each machine), integer values that are valid pointers are only 0 or 1 (since the memories are two elements long), and stacks do not need to be large either.

Consider, for instance, a generator for stacks (of a given length `n`), which can be empty (`Mty`), cells that store a tagged integer (`Cons`), or specially marked stack frames that store a program counter to be used by a future `Return` instruction (`RetCons`); `gen_atom` produces mostly in-bounds tagged integers.

```
Fixpoint gen_stack (n : nat) : G Stack :=
  match n with
    | O => returnGen Mty
    | S n' =>
      freq [ (10, liftGen2 Cons gen_atom (gen_stack n'))
           ; (4,  liftGen2 RetCons gen_atom (gen_stack n'))
           ]
  end.
```

The behavior of this generator can be described by a simple inductive predicate, where good_atom describes the behavior of gen_atom.

```
Inductive good_stack : nat -> Stack -> Prop :=
| GoodStackMty  : good_stack 0 Mty
| GoodStackCons : forall n a s ,
    good_atom a  -> good_stack n s -> good_stack (S n) (a  :: s)
| GoodStackRet  : forall n pc s,
    good_atom pc -> good_stack n s -> good_stack (S n) (RetCons pc s).
```

Finally, we can achieve the same distribution with a weight annotation before deriving generators.

```
  QuickChickWeights [(GoodStackCons, 10); (GoodStackRet, 4)].
  Derive ArbitrarySizedSuchThat for (fun s => good_stack n s).
```

The implicit assumptions for single-state generators are encoded in inductive predicates, and the indistinguishability relation is used to derive variation generators.

We tested the single-step noninterference property 10000 times using both the handwritten and the derived generators. Our derived generators were 1.75× slower than the handwritten ones, while both generators uncovered all mutants successfully. To ensure both generators yield similar distributions of inputs, we used QuickChick's `collect` to determine the number of times each instruction was generated during those 10000 tests (as this was the metric that was used to fine-tune the handwritten generators in the first place).

The observed 1.75× slowdown is mostly due to the added overhead of local backtracking and extraneous matches like the one in the goodTree example of Section 3. A few local optimizations (like pulling a match outside of a call to `backtrack`) could further improve on our performance, but would require additional work to produce the corresponding proof terms. Still, this overhead is much better than the order-of-magnitude overhead of interpreted approaches like Luck (Lampropoulos et al. 2017). Finally, what we gain in return for this loss in performance is that the declarative nature of the inductive predicates exposes exactly what assumptions are made about the generated domain, while the produced proofs guarantee completeness for that domain.

8 RELATED WORK

Narrowing-based approaches. The most closely related works, narrowing-based approaches, have already been briefly discussed in the introduction. Such approaches have given rise to tools for many languages, including Haskell (Claessen et al. 2014) and Racket (Fetscher et al. 2015), as well as domain-specific languages, like UDITA (Gligoric et al. 2010) and Luck (Lampropoulos et al. 2017). All of these artifacts successfully adapt variants of narrowing to generate values satisfying preconditions.In this paper, we build upon their success, adapting narrowing for Coq's inductive relations, showing how to produce Coq generators getting rid of interpretation overheads, and producing proofs of the generators correctness in the process.

Smart enumeration approaches. Another closely related line of work deals with *enumeration* for data satisfying invariants. Perhaps closest to us is the work in the context of Isabelle's QuickCheck (Bulwahn 2012a,b). There, Bulwahn targets a similar simply-typed subset of Isabelle producing ML enumerators. A different enumeration approach, based on laziness, is taken for Haskell's Lazy SmallCheck (Runciman et al. 2008), as well as Scala's SciFe (Kuraj et al. 2015). Laziness conceptually mimics narrowing by delaying the instantiation of variables when possible. On the other hand, (Fischer and Kuchen 2007) and (Christiansen and Fischer 2008) leverage the built-in narrowing mechanism of the functional logic programming language Curry (Hanus et al. 1995) to enumerate data satisfying invariants. While enumeration based testing can be very successful in various domains, we turn to random testing as the complexity of our target applications—like, for example, well typed lambda terms—makes enumeration intractable.

SMT-based approaches. An alternative approach to generating inputs satisfying a precondition P is to translate P into logic and then use an SMT solver. Such a translation has been performed a few times (Carlier et al. 2013; Gotlieb 2009; Seidel et al. 2015)). The most recent and efficient one, Target (Seidel et al. 2015), targets Liquid Haskell preconditions in the form of refinement types. While Target outperforms Lazy SmallCheck and similar tools in a lot of cases, Luck (Lampropoulos et al. 2017) shows that narrowing can still perform better on occasion. Moreover, the complexity of the translation leaves little room, if any, for controlling the distribution of generated inputs, unlike in QuickChick-derived generators where users can leverage Luck-style annotations to control `backtrack` weights.

Inductive to Executable Specifications. At a high level, the algorithm described in Section 4 has similarities to earlier attempts at extracting executable specifications from inductive ones (Delahaye et al. 2007; Tollitte et al. 2012) in the Coq proof assistant. In principle, we could use the algorithm described in this section to obtain a similar transformation. Consider for example, an inductive predicate `P : A -> B -> C -> Prop`. If we transform it to a predicate `P' : A -> B -> C -> unit -> Prop` by adding `()` as an additional argument at every occurrence of `P`, we could ask our algorithm to generate `x` such that `P' a b c x` holds for all `a`, `b`, and `c`. We would then essentially obtain a partial decision procedure for `P`, based on whether the generator returns `Some` or `None`. In fact, our algorithm can be seen as a generalization of their approach as the derived decision procedures are equivalent (modulo size) for the class of inductive datatypes they handle that yields deterministic functional programs.

9 CONCLUSION AND FUTURE WORK

We have presented a narrowing-based algorithm for compiling dependently-typed inductive relations into generators for random data structures satisfying these relations, together with correctness proofs. We implemented it in the Coq proof assistant and evaluated its applicability by automatically deriving generators to test the majority of theorems in *Software Foundations*.

In the future, we aim to extend our algorithm to a larger class of inductive definitions. For example, incorporating function symbols is straightforward: simply treat functions as black boxes, instantiating all of their arguments before treating the result as a *fixed* range. For statically known functions, we could also leverage Coq's open term reduction to try to simplify function calls into constructor terms. Finally, it would be possible to adapt the established narrowing approaches for functional programs to meaningfully instantiate unknown function arguments against a known result pattern, just like in Luck (Lampropoulos et al. 2017).

We also want to see if our algorithm can be adapted to derive decidability instances for specifications in `Prop`, allowing for immediate, fully automatic testing feedback. We are also interested in *shrinkers* for constrained data, to complete the property-based testing ecosystem for Coq.

ACKNOWLEDGMENTS

We are grateful to Maxime Dénès, Cătălin Hriţcu, John Hughes, George Karachalias, Michal Palka, Antal Spector-Zabusky, the CLA workshop community, and the Penn PLClub for their useful comments. This material is based upon work supported by the National Science Foundation under Grant No. 1421243 (*Random Testing for Language Design*) and Grant No. 1521523 (*Expeditions in Computing: The Science of Deep Specification*). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Sergio Antoy. 2000. A Needed Narrowing Strategy. In *Journal of the ACM*, Vol. 47. ACM Press, 776–822. https://www.informatik.uni-kiel.de/~mh/papers/JACM00.pdf

Thomas Arts, Laura M. Castro, and John Hughes. 2008. Testing Erlang Data Types with QuviQ QuickCheck. In *7th ACM SIGPLAN Workshop on Erlang*. ACM, 1–8. DOI: http://dx.doi.org/10.1145/1411273.1411275

Lukas Bulwahn. 2012a. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *2nd International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science)*, Vol. 7679. Springer, 92–108. https://www.irisa.fr/celtique/genet/ACF/BiblioIsabelle/quickcheckNew.pdf

Lukas Bulwahn. 2012b. Smart Testing of Functional Programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science)*, Vol. 7180. Springer, 153–167. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.229.1307&rep=rep1&type=pdf

Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. 2013. FocalTest: A Constraint Programming Approach for Property-Based Testing. In *Software and Data Technologies (Communications in Computer and Information Science)*, Vol. 170. Springer, 140–155. DOI: http://dx.doi.org/10.1007/978-3-642-29578-2_9

Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. 2011. Integrating Testing and Interactive Theorem Proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications (EPTCS)*, Vol. 70. 4–19. http://arxiv.org/abs/1105.4394

Jan Christiansen and Sebastian Fischer. 2008. EasyCheck – Test Data for Free. In *9th International Symposium on Functional and Logic Programming (FLOPS) (Lecture Notes in Computer Science)*, Vol. 4989. Springer, 322–336. http://www-ps.informatik.uni-kiel.de/~sebf/data/pub/flops08.pdf

Koen Claessen, Jonas Duregård, and Michal H. Palka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming (Lecture Notes in Computer Science)*, Vol. 8475. Springer, 18–34. DOI: http://dx.doi.org/10.1007/978-3-319-07151-0_2

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 268–279. http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. 2007. Extracting Purely Functional Contents from Logical Inductive Types. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science)*, Vol. 4732. Springer, 70–85. http://cedric.cnam.fr/~delahaye/papers/pred-exec%20(TPHOLs'07).pdf

Maxime Dénès, Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2014. QuickChick: Property-based testing for Coq. The Coq Workshop. (July 2014). http://prosecco.gforge.inria.fr/personal/hritcu/talks/coq6_submission_4.pdf

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2003. Combining Testing and Proving in Dependent Type Theory. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (Lecture Notes in Computer Science)*, Vol. 2758. Springer, 188–203. http://www.cse.chalmers.se/~peterd/papers/Testing_Proving.pdf

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2004. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology* 46, 15 (2004), 1011–1025. http://www.cse.chalmers.se/~peterd/papers/TestingModelChecking.pdf

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *24th European Symposium on Programming (Lecture Notes in Computer Science)*, Vol. 9032. Springer, 383–405. http://users.eecs.northwestern.edu/~baf111/random-judgments/

Sebastian Fischer and Herbert Kuchen. 2007. Systematic generation of glass-box test cases for functional logic programs. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, 63–74. http://www-ps.informatik.uni-kiel.de/~sebf/pub/ppdp07.html

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *32nd ACM/IEEE International Conference on Software Engineering*. ACM, 225–234. DOI: http://dx.doi.org/10.1145/1806799.1806835

Arnaud Gotlieb. 2009. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *ICST 2009, Second International Conference on Software Testing Verification and Validation, 1-4 April 2009, Denver, Colorado, USA*. 151–160. DOI: http://dx.doi.org/10.1109/ICST.2009.10

M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. 1995. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*. 95–107. http://www.math.rug.nl/~piter/KR/hanus95curry.pdf

Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 455–468. http://prosecco.gforge.inria.fr/personal/hritcu/publications/testing-noninterference-icfp2013.pdf

Cătălin Hriţcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. *Journal of Functional Programming (JFP); Special issue for ICFP 2013* 26 (April 2016), e4 (62 pages). DOI: http://dx.doi.org/10.1017/S0956796816000058 Technical Report available as arXiv:1409.0393.

John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL) (Lecture Notes in Computer Science)*, Vol. 4354. Springer, 1–32. http://people.inf.elte.hu/center/fulltext.pdf

Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. http://crest.cs.ucl.ac.uk/fileadmin/crest/sebasepaper/JiaH10.pdf

Ivan Kuraj and Viktor Kuncak. 2014. SciFe: Scala framework for efficient enumeration of data structures with invariants. In *Proceedings of the Fifth Annual Scala Workshop*. ACM, 45–49. DOI: http://dx.doi.org/10.1145/2637647.2637655

Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. 2015. Programming with Enumerable Sets of Structures. In *OOPSLA*. http://lara.epfl.ch/~kuncak/papers/KurajETAL15ProgrammingEnumerableSetsStructures.pdf

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 114–129. http://dl.acm.org/citation.cfm?id=3009868

Fredrik Lindblad. 2007. Property Directed Generation of First-Order Test Data. In *8th Symposium on Trends in Functional Programming (Trends in Functional Programming)*, Vol. 8. Intellect, 105–123. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.2439&rep=rep1&type=pdf

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: feedback-directed random testing for Java. In *22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems And Applications (OOPSLA)*. ACM, 815–816. DOI:http://dx.doi.org/10.1145/1297846.1297902

Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 91–97. DOI:http://dx.doi.org/10.1145/1982595.1982615

Manolis Papadakis and Konstantinos F. Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. 39–50. DOI:http://dx.doi.org/10.1145/2034654.2034663

Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *6th International Conference on Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 325–343. http://prosecco.gforge.inria.fr/personal/hritcu/publications/foundational-pbt.pdf

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2016. *Software Foundations*. Electronic textbook, Version 4.0 beta. https://www.cis.upenn.edu/~bcpierce/sf/sf-4.0/index.html

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science)*, Bernhard Steffen (Ed.), Vol. 1384. Springer, 151–166. DOI:http://dx.doi.org/10.1007/BFb0054170

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *1st ACM SIGPLAN Symposium on Haskell*. ACM, 37–48. http://www.cs.york.ac.uk/fp/smallcheck/smallcheck.pdf

Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 812–836. DOI:http://dx.doi.org/10.1007/978-3-662-46669-8_33

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*. Springer-Verlag, Berlin, Heidelberg, 278–293. DOI:http://dx.doi.org/10.1007/978-3-540-71067-7_23

Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. 2012. Producing Certified Functional Code from Inductive Specifications. In *Second International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science)*, Vol. 7679. Springer. http://cedric.cnam.fr/~delahaye/papers/relext-coq%20%28CPP%2712%29.pdf

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76. DOI:http://dx.doi.org/10.1145/75277.75283

Li-yao Xia. 2017. `generic-random`: Generic, Customizable Arbitrary Instances. http://hackage.haskell.org/package/generic-random. (10 April 2017).