The Search for Constrained Random Generators

ANONYMOUS AUTHOR(S)

Among the most significant challenges in property-based testing (PBT) is the *constrained random generation problem*: given a predicate on program values, randomly sample from the set of all values satisfying that predicate, and only those values. Efficient solutions to this problem are critical for effective testing, since the executable specifications used by PBT often have preconditions that input values must satisfy in order to be valid test cases, and satisfying values are often sparsely distributed.

We present a novel approach to the constrained random generation problem using ideas from deductive program synthesis. We propose a set of synthesis rules, based on a denotational semantics of generators, that give rise to an automatic procedure for synthesizing correct generators. We deal with recursive predicates by rewriting them as catamorphisms (i.e., folds) and then matching with appropriate anamorphisms (i.e., unfolds); this is theoretically simpler than other approaches to synthesis for recursive functions, yet still extremely expressive. Our implementation, Palamedes, is an extensible library for the Lean theorem prover.

1 INTRODUCTION

Property-based testing (PBT) [11] bridges the gap between traditional testing and heavier-weight formal methods [56] by allowing developers to automatically test software systems against formal specifications. PBT is used to find bugs in a wide range of real-world software [1, 3, 4, 7], but significant work remains. A key challenge is the *constrained random generation problem*.

Consider a classic example of the kind of property one might test with PBT:

$$\forall\, x\ t,\ \mathsf{isBST}(t) \implies \mathsf{isBST}(\mathsf{insert}(x,\ t))$$

This says that, if a tree t is a valid binary search tree (BST), then inserting a new value x into t yields another valid BST. To test this property, a PBT framework will use a program called a *generator* to randomly sample hundreds or thousands of values for x and t and check that the property holds for each pair of values. However, the property is vacuously true if t is not a valid BST to start with, so, unless the random generation procedure is not carefully designed, most generated trees will be discarded as invalid and only a few will actually be used to test the insert function. This is the main motivation for the constrained random generation problem: to test a conditional property effectively, we need a way to randomly sample from the set of *all and only* the values that satisfy its precondition.

This problem has been studied extensively by programming languages researchers over the years [10, 17, 27, 51], with many proposed solutions that address it to some degree. But, as of 2024, a study on the practical usability of PBT [16] still cited the availability of generators as a key challenge for PBT adoption. The key issue is speed: this study showed that many developers run PBT often, so they need it to be fast. But most of the established approaches to constrained generation perform some kind of combinatorial search to generate valid inputs (e.g., rejection sampling, calling SMT solvers, or using feedback from code coverage), which slows down generation significantly.

We propose a different approach. Rather than searching *during* generation, we search ahead of time, using a *deductive synthesis* algorithm to search for generators that are guaranteed by construction to be fast and to produce valid values. This synthesis algorithm can run "offline," before testing time, allowing for faster and more effective testing. The algorithm is based on a denotational semantics that captures the values that a given generator can generate. If we want a generator for a property whose precondition is φ, we compute a generator g satisfying

$$\forall\, a,\ a \in [\![g]\!] \iff \varphi(a),$$

2025. 2475-1421/2025/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

where $[\![g]\!]$ is the set of values the generator can generate. In other words, a can be generated by g if and only if $\varphi(a)$ holds. Inspired by systems like Synquid and SuSLik [43, 44], our synthesis algorithm actually builds a *proof* that there exists an appropriate g for a given φ by applying a series of proof rules, then extracts a proof witness that embodies g in executable form.

Our basic proof rules are roughly one-to-one with common *generator combinators*—the core functions used to build generators by hand. To construct recursive generators for inductively defined data structures, we use *recursion schemes*; concretely, we observe that many predicates that can be represented as a fold (or catamorpishm) have an associated generator represented as an unfold (or anamorphism). The upshot is that we can deal with recursive predicates without relying on cyclic proofs [23], instead using pre-derived induction principles.

We implement our synthesis algorithm as a library in the Lean theorem prover that builds Lean generators from predicates expressed as Lean functions. Working in a theorem prover brings several benefits. First, synthesized generators come with mechanized proofs of correctness, guaranteeing that any generator produced by our algorithm produces exactly the desired set of values. Next, working in Lean means the implementation of the synthesis procedure can be relatively simple—in fact, we use a popular proof search tactic, Aesop [30], to do much of the heavy lifting. Finally, extending the algorithm with new primitives or search tactics is as simple as proving a lemma or writing a macro. The synthesizer is already quite powerful—it can, for example, synthesize generators for BSTs and well-typed simply-typed lambda calculus (STLC) programs—and the set of predicates it can handle can continue to grow modularly.

Section 2 below offers more concrete motivation and background, along with a preview of our approach. The remainder of the paper presents the following contributions:

- We propose a system of deductive synthesis rules for correct generators (Section 3). These rules are proven correct relative to a denotational semantics of generators.
- We extend the resulting synthesis algorithm to work for generators of recursive data types, leveraging intuitions from the literature on recursion schemes (Section 4) to establish a relationship between predicates written as folds and generators written as unfolds.
- We describe Palamedes, an implementation of our synthesis procedure embedded as a library in the Lean proof assistant (Section 5). This implementation strategy achieves performant synthesis that also maintains mechanized proofs of correctness, while borrowing key parts of the algorithm from Lean's existing infrastructure.
- We evaluate our approach on a suite of case studies, demonstrating that the synthesis procedure produces generators comparable to handwritten ones for a wide range of predicates (Section 6). Our most complex case studies—generators for binary search trees, AVL trees, and well-typed STLC terms—have been used as PBT benchmarks for decades.

We conclude with a discussion of limitations (Section 7), related work (Section 8), and future work (Section 9).

2 MOTIVATION

We begin by explaining the constrained random generation problem (Section 2.1), reviewing current approaches to the problem (Section 2.2), and motivating a refined version (Section 2.3)—the constrained generator *synthesis* problem—that we address in the rest of the paper.

2.1 Fast Generators for Property-Based Testing

Property-based testing uses random test data to validate executable program specifications. In this paper, our focus is on the *generators* that produce the random test inputs used to exercise the system under test, which are critical to performant and effective testing. For example, when testing the BST property in Section 1, a developer might write a generator like the one in Figure 1.[1]

```
def genBST lo hi :=
  if lo > hi then
    pure leaf
  else
    pick
      (pure leaf)
      (do
        let x <- choose lo hi
        let l <- genBST lo (x - 1)
        let r <- genBST (x + 1) hi
        pure (node l x r))
```

Fig. 1. A handwritten generator for binary search trees.

This generator produces random BSTs with values in a given range. In the case where the range is empty, `pure` creates a constant generator that always returns `leaf`. Otherwise, `pick` is used to make a choice: either generate a `leaf` or generate a `node` by selecting a value in the appropriate range and recursively generating subtrees with truncated ranges. The **do** notation sequences generators by sampling from a generator, binding the sampled value to a variable, and continuing as another generator.

While generators like this are familiar to PBT experts, novices might struggle to come up with them from scratch. Moreover, a recent study of PBT users [16] found that even experts, who can in principle write effective generators like the one above, still see writing generators as a distraction from other testing tasks.

What makes an effective generator? We can start with two key properties:

(1) **Soundness** The generator should produce *only* values that satisfy a given validity predicate.
(2) **Completeness** The generator should be able to produce *any* value that satisfies the validity predicate. This ensures that the generator does not miss parts of the input space.[2]

We can now phrase the constrained random generation problem more formally:

Definition 2.1 (Constrained Random Generation Problem)**.** Given a predicate φ, sample random values in a way that is sound and complete with respect to φ.

Solving this problem—in particular, finding fast solutions—is critical if we want to build PBT tools that are convenient enough to be usable by mainstream developers and powerful enough to find bugs in mainstream software.

2.2 Search-Based Approaches to Generation

Many approaches to the constrained random generation problem have been proposed over the years, most based around some kind of search procedure. There are a few notable exceptions [26, 28] (we delay comparisons with these to Section 8, where we can go into more technical detail), but most of the available search procedures start with a naïve, complete-but-not-sound PBT generator (e.g., derived from type information [34, 58]) and prune its generation space "online," during the generation process, to achieve soundness. The simplest form of pruning is *rejection sampling*—discarding any invalid values and retrying generation—but this results in poor performance for

[1]An alternative strategy for generating BSTs, in particular, is to generate a list of numbers and insert those numbers one by one into an empty BST. This approach is simple and effective, but it does not generalize to other predicates and data structures. We focus on generators that yield constrained data structures by following the structure of a validity predicate.

[2]Readers familiar with the PBT literature might expect us to say something about the *distribution* of a complete generator, but (aside from this aside) we are not going to. We treat generators as nondeterministic programs, focusing on the set of values that they can produce and ignoring the probabilities with which they produce them. This is not because probabilities are unimportant—they can have a huge impact on testing performance—but because they can be handled separately. Recent work [54] has shown that, once we have a generator, its internal weights can be tuned by an external process to achieve various desirable distributional qualities; we defer distributional concerns to such processes. See Section 9 for more.

even moderately sparse predicates. More advanced pruning mechanisms include laziness [10], Brzozowski derivatives [17], reinforcement learning [47], and constraint solving [49, 51]. Other approaches try to search for valid inputs "from scratch," for example with the help of a large language model [57].

There is something unsatisfying about all of these search-based approaches: the `genBST` generator above does no searching at all! It produces valid inputs by construction, every time, essentially as quickly as possible. This should lead us to wonder whether we could automatically build other generators that, like `genBST`, avoid generation-time search...

2.3 The Constrained Generator Synthesis Problem

We propose a refinement of the constrained random generation problem that focuses specifically on building correct generators directly, rather than relying on generation-time search:

Definition 2.2 (Constrained Generator Synthesis Problem)**.** Given a predicate φ, synthesize an efficient generator g that is sound and complete with respect to φ.

The difference between this and Definition 2.1 is subtle but important: we are specifically interested in solutions that synthesize an efficient generator first and then use it to sample valid values; solutions that actively search for valid values during generation are excluded. (We discuss what we mean by "efficient" in the next section.) This version of the problem is harder, but solutions have the potential to be much more effective for PBT users, since generators are typically written once and then run many times.

3 DEDUCTIVE SYNTHESIS FOR GENERATORS

We propose a novel approach to the constrained generator synthesis problem. We start by introducing our representation of generators and some key definitions (Section 3.1); next we present our core deductive synthesis rules for constructing generators (Section 3.2) and introduce some handy base generators (Section 3.3); finally, we discuss an optimization procedure that can be applied to generators after synthesis (Section 3.4).

The presentation in this section is phrased in terms of an unspecified ambient dependent type theory; in Section 5 we make the definitions and proofs concrete in the Lean proof assistant.

3.1 Generator Representation

Generators for values of type `a` are typically represented as sampling functions of type `Seed -> a`, but we opt for a representation with a bit more flexibility. Taking inspiration from work on *free generators* [17], we represent generators as data structures that can be interpreted in multiple ways, including as sampling functions. These structures belong to the following inductive data type:

$$
\begin{array}{l}
\mathsf{inductive\ Gen\ where} \\
\quad \mathsf{pure} : \alpha \to \mathsf{Gen}\ \alpha \\
\quad \mathsf{bind} : \mathsf{Gen}\ \beta \to (\beta \to \mathsf{Gen}\ \alpha) \to \mathsf{Gen}\ \alpha \\
\quad \mathsf{pick} : \mathsf{Gen}\ \alpha \to \mathsf{Gen}\ \alpha \to \mathsf{Gen}\ \alpha \\
\quad \mathsf{indexed} : (\mathbb{N} \to \mathsf{Gen}\ (\mathsf{Option}\ \alpha)) \to \mathsf{Gen}\ \alpha \\
\quad \mathsf{assume} : (b : \mathbb{B}) \to (b = \mathsf{true} \to \mathsf{Gen}\ \alpha) \to \mathsf{Gen}\ \alpha
\end{array}
$$

We often write bind with the infix notation $\gg\!=$.

Each of these constructors has a standard interpretation as a procedure for sampling values, discussed in more detail in Section 5. The pure constructor represents a constant generator that always produces the same value. The $\gg\!=$ constructor represents sequencing of generators, sampling from one and passing the sampled value to a function producing another. The pick constructor represents a choice between generators. The assume constructor represents a partial generator. Intuitively, it checks a boolean condition; if true, it simply calls its argument, but if false it generates nothing. When interpreted as a sampling function, assume is a partial function that may need to be retried or backtracked around. The indexed constructor represents an infinite family of generators, indexed by natural numbers. (In lazy languages like Haskell this constructor is not necessary. But since we will be working in Lean, which is strict, we need it to be able to represent generators of infinite sets—e.g., a generator of all natural numbers.)

Definition 3.1. The *support* of a generator g is the set of values that g can produce, written $[\![g]\!]$. Support is defined as follows:

$$\begin{aligned}
a \in [\![\mathsf{pure}\ a']\!] &\iff a = a' \\
a \in [\![x \gg\!= f]\!] &\iff \exists\, a',\ a' \in [\![x]\!] \land a \in [\![f\ a']\!] \\
a \in [\![\mathsf{pick}\ x\ y]\!] &\iff a \in [\![x]\!] \lor a \in [\![y]\!] \\
a \in [\![\mathsf{indexed}\ f]\!] &\iff \exists\, n,\ \mathsf{some}\ a \in [\![f\ n]\!] \\
a \in [\![\mathsf{assume}\ b\ \mathsf{in}\ x]\!] &\iff b = \mathsf{true} \land a \in [\![x]\!]
\end{aligned}$$

This follows the intuition given above and agrees with the denotational semantics of generators proposed in prior work [40].

Example 3.1 (Generator for Natural Numbers). The following generator uses all but one of the above constructors to generate the set of all natural numbers:

```
def arbNat : Gen ℕ :=
  let rec go (fuel : ℕ) : Gen (Option ℕ) :=
    match fuel with
    | 0 ⇒ pure none
    | fuel′ + 1 ⇒
      pick
        (pure (some 0))
        (go fuel′ >>= λ on′ ⇒
          match on′ with | none ⇒ pure none | some n′ ⇒ pure (some (1 + n′)))
  indexed go
```

It defines a recursive function go that takes some fuel and produces a potentially failing generator of natural numbers. If the fuel has run out, the generator fails with none. Otherwise, the generator makes a random choice between returning 0 and returning $1+n'$, where n' is generated by recursively calling go. The outer indexed turns this indexed family of partial generators into a total generator. When interpreted, indexed gives its argument an arbitrarily large amount of fuel, so we need only worry about the "happy path."

The support of this generator is precisely ℕ (as we prove in our Lean development).

Example 3.2 (Backtracking Generator). While our ultimate goal is to avoid generators that search during generation, we need such generators to be representable in our language (we will see why later, when synthesizing generators for BSTs). Our synthesis procedure works by first producing generators that need to backtrack and then attempting to optimize them.

Here is an example of a generator that backtracks due to assume:

```
def genBacktrack := pick (pure 1) (assume false in pure 2)
```

The support of this generator is $\{1\}$—that is to say, when it generates a value, it always generates 1—but it will sometimes choose the right side of the pick and fail.

We can use support to formally define what it means for a generator to be correct.

Definition 3.2 (Correctness)**.** A generator g is *correct* with respect to a predicate φ if it is both *sound*, i.e., $\forall\, a,\ a \in [\![g]\!] \implies \varphi(a)$, and *complete*, i.e., $\forall\, a,\ \varphi(a) \implies a \in [\![g]\!]$.

Notation 3.1 (Correct Generator)**.** The type of correct generators with respect to a predicate φ on type α is denoted $\mathsf{Gen}_\alpha\ \varphi$. The type of the generated value becomes a subscript; we may leave it off if it is clear from context.

Example 3.3. The arbNat generator above can be given the type $\mathsf{Gen}_{\mathbb{N}}\ (\lambda\, n \Rightarrow \top)$; backtracks has type $\mathsf{Gen}_{\mathbb{N}}\ (\lambda\, n \Rightarrow n = 1)$.

We now return to the notion of "efficiency" that we postponed in §2.3. Our gold standard for generators is exemplified by `genBST`—generators that are like the ones written by expert users to produce valid inputs by construction. The synthesis procedure we describe over the next few sections meets this standard in most cases, as we demonstrate in Section 6 by comparing synthesized generators with user-written ones, but there are situations in which it is not completely successful. In particular, it sometimes produces generators that use the assume constructor in ways that require the generator to backtrack at run-time.

To distinguish these correct-but-suboptimal generators from the more performant ones we aim to synthesize as often as possible, we define the following notion:

Definition 3.3 (Assume Freedom)**.** A generator g is *assume-free* iff it does not mention the assume constructor (including in sub-generators that it calls).

Our case studies in Section 6 are clearly marked to show which are assume-free; all are correct by construction.

3.2 Core Synthesis Algorithm

We now outline a semi-algorithm to solve the correct generator synthesis problem. The algorithm uses *deductive program synthesis*, constructing a generator by working backwards from the structure of the predicate. It creates a proof that witnesses the generator's correctness and builds the generator itself *en passant*. This approach to synthesis can also be found in systems like SuSLik [44] and Synquid [43].

Concretely, we start with the statement

$$\frac{}{\Gamma \vdash ? : \mathsf{Gen}_\alpha\ P}\ ?$$

and successively refine the generator by applying a series of *synthesis rules* to build a complete derivation.

Pure and Pick. Here are our first two basic synthesis rules:

$$\frac{\Gamma \vdash a' : \alpha}{\Gamma \vdash \mathsf{pure}\ a' : \mathsf{Gen}_\alpha\ (\lambda\, a \Rightarrow a = a')}\ \textsc{S-Pure} \qquad \frac{\Gamma \vdash g_1 : \mathsf{Gen}_\alpha\ P \qquad \Gamma \vdash g_2 : \mathsf{Gen}_\alpha\ Q}{\Gamma \vdash \mathsf{pick}\ g_1\ g_2 : \mathsf{Gen}_\alpha\ (\lambda\, a \Rightarrow P\ a \vee Q\ a)}\ \textsc{S-Pick}$$

The former says that we can synthesize a value that is equal to a constant using pure, and the latter says that we can synthesize for a disjunction by using pick. We can use these rules to synthesize a generator for the predicate $\lambda\, a \Rightarrow a = 1 \vee a = 2$, as shown in Figure 2. Note how the generator at the root of the derivation is refined as rules are added.

While the core of the proof tree in Figure 2 is focused on the synthesis rules, there are other things going on. In particular, when applying S-Pure on the left, we also need to prove that $\cdot \vdash 1 : \mathbb{N}$. In this case the proof is trivial, but in general the synthesizer may need to be able to discharge

$$\dfrac{}{\cdot \vdash \,?: \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1 \lor a = 2)}\ ?$$

apply S-Pick

$$\dfrac{\dfrac{}{\cdot \vdash \,?x : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1)}\ ? \qquad \dfrac{}{\cdot \vdash \,?y : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 2)}\ ?}{\cdot \vdash \mathsf{pick}\ ?x\ ?y : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1 \lor a = 2)}\ \textsc{S-Pick}$$

apply S-Pure **on the left**

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1 : \mathbb{N}}\ \Downarrow}{\cdot \vdash \mathsf{pure}\ 1 : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1)}\ \textsc{S-Pure} \qquad \dfrac{}{\cdot \vdash \,?y : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 2)}\ ?}{\cdot \vdash \mathsf{pick}\ (\mathsf{pure}\ 1)\ ?y : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1 \lor a = 2)}\ \textsc{S-Pick}$$

apply S-Pure **on the right**

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 1 : \mathbb{N}}\ \Downarrow}{\cdot \vdash \mathsf{pure}\ 1 : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1)}\ \textsc{S-Pure} \qquad \dfrac{\dfrac{}{\cdot \vdash 2 : \mathbb{N}}\ \Downarrow}{\cdot \vdash \mathsf{pure}\ 2 : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 2)}\ \textsc{S-Pure}}{\cdot \vdash \mathsf{pick}\ (\mathsf{pure}\ 1)\ (\mathsf{pure}\ 2) : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1 \lor 1 = 2)}\ \textsc{S-Pick}$$

Fig. 2. A step-by-step example of synthesizing a generator.

certain nontrivial theorems automatically. In §5 we discuss the specifics of this process; for now we assert that any rule labeled "$\Downarrow$" can be discharged automatically.

Sometimes the synthesis rules do not apply directly to the goal as stated. For example consider this goal:

$$\dfrac{}{\cdot \vdash \,?: \mathsf{Gen}\ (\lambda\ a \Rightarrow 1 = a)}\ ?$$

This is obviously equivalent to a goal that we know how to deal with (i.e., with $a = 1$), but it is not syntactically the same. In these cases, we may need to apply the *conversion rule*:

$$\dfrac{\dfrac{}{\Gamma \vdash P = Q}\ \Downarrow \qquad \Gamma \vdash g : \mathsf{Gen}\ Q}{\Gamma \vdash g : \mathsf{Gen}\ P}\ \textsc{Convert}$$

The resulting derivation looks like this:

$$\dfrac{\dfrac{}{\cdot \vdash (\lambda\ a \Rightarrow a = 1) = (\lambda\ a \Rightarrow 1 = a)}\ \Downarrow \qquad \dfrac{}{\cdot \vdash \mathsf{pure}\ 1 : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = 1)}\ \textsc{S-Pure}}{\cdot \vdash \mathsf{pure}\ 1 : \mathsf{Gen}\ (\lambda\ a \Rightarrow 1 = a)}\ \textsc{Convert}$$

Assumptions and Functions. If the generator we need is already available in the typing context Γ, we can just use it.

$$\dfrac{(x : \mathsf{Gen}_\alpha\ P) \in \Gamma}{\Gamma \vdash x : \mathsf{Gen}_\alpha\ P}\ \textsc{S-Assumption}$$

When the synthesis goal is a function returning a generator, we can apply an introduction rule.

$$\frac{b{:}\beta, \Gamma \vdash g : \mathsf{Gen}_\alpha\ P}{\Gamma \vdash (\lambda\ b \Rightarrow g) : (b : \beta) \to \mathsf{Gen}_\alpha\ P}\ \textsc{S-Intro}$$

This rule says that if we can produce an appropriate generator given $b{:}\beta$ in the context, then we can produce a (dependent) function from β to that generator.

We also need a special case for dealing with functions that take tuples as arguments, which appear frequently as a result of our rules for recursive functions (see Section 4):

$$\frac{\Gamma \vdash f : (b : \beta) \to (c : \gamma) \to \mathsf{Gen}_\alpha\ (P\ b\ c)}{\Gamma \vdash (\lambda\ (b, c) \Rightarrow f\ b\ c) : (p : \beta \times \gamma) \to \mathsf{Gen}_\alpha\ (P\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p))}\ \textsc{S-Uncurry}$$

In words, this rule says that we are free to synthesize a curried function and then uncurry it, if the goal is to produce a function with a tuple argument.

Bind. We can define a synthesis rule for composing generators by analogy with the definition of support for bind:

$$\frac{\Gamma \vdash g : \mathsf{Gen}_{\alpha'}\ P \qquad \Gamma \vdash f : (a' : \alpha') \to \mathsf{Gen}_\alpha\ (Q\ a')}{\Gamma \vdash g \gg\!= f : \mathsf{Gen}_\alpha\ (\lambda\ a \Rightarrow \exists\ (a' : \alpha'),\ P\ a' \land Q\ a'\ a)}\ \textsc{S-Bind}$$

The goal of this rule requires that the predicate we want to generate for is an existential statement with two conjuncts; P should be some statement constraining a value a', and then Q should constrain the final value, a, given a particular a'. A generator for a predicate of this form looks like a generator x of a's satisfying P plus a function f that takes an a' and produces a generator for as satisfying the predicate $Q\ a'$.

We can use $\gg\!=$ to chain generators together:

$$\frac{\dfrac{\cdots}{\Gamma \vdash \cdots : \mathsf{Gen}\ (\lambda\ j \Rightarrow j = 1 \lor j = 2)} \qquad \dfrac{\dfrac{}{j{:}\mathbb{N}, \Gamma \vdash \mathsf{pure}\ (j+3) : \mathsf{Gen}\ (\lambda\ i \Rightarrow i = j + 3)}}{\Gamma \vdash \lambda\ j \Rightarrow \mathsf{pure}\ (j+3) : j{:}\mathbb{N} \to \mathsf{Gen}\ (\lambda\ i \Rightarrow i = j + 3)}}{\Gamma \vdash \mathsf{pick}\ (\mathsf{pure}\ 1)\ (\mathsf{pure}\ 2) \gg\!= (\lambda\ j \Rightarrow \mathsf{pure}\ (j+3)) : \mathsf{Gen}\ (\lambda\ i \Rightarrow \exists\ j,\ (j = 1 \lor j = 2) \land i = j + 3)}$$

The goal here is to generate an i that is equal to $j+3$, where j is constrained to be either 1 or 2. To synthesize an appropriate generator, we use S-Bind, which gives two sub-goals. On the left, the goal is to synthesize a generator for j, which we complete as above. On the right, we apply S-Intro followed by S-Pure to produce the continuation of the bind. The final generator generates 1 or 2 and then adds 3.

With the help of the Convert rule, S-Bind can apply in a wide range of situations. For example, the predicate `isSome x` does not contain explicit existential quantification, but it is equivalent to $\exists\ a,\ \top \land x = \mathsf{some}\ a$, so the synthesizer can Convert the predicate and then apply S-Bind. We discuss the logical manipulations that are applied to predicates in Section 5.2.

Case Splitting. If a generator needs to do different things based on the value of a variable in the context, it can use one of the following rules to synthesize pattern matches.

$$\frac{\Gamma \vdash g_t : \mathsf{Gen}_\alpha\ (P\ \mathsf{true}) \qquad \Gamma \vdash g_f : \mathsf{Gen}_\alpha\ (P\ \mathsf{false})}{b{:}\mathbb{B}, \Gamma \vdash \mathsf{match}\ b\ \mathsf{with}\ \mid \mathsf{true} \Rightarrow g_t \mid \mathsf{false} \Rightarrow g_f : \mathsf{Gen}_\alpha\ (P\ b)}\ \textsc{S-SplitBool}$$

$$\frac{\Gamma \vdash g_z : \mathsf{Gen}_\alpha\ (P\ 0) \qquad n'{:}\mathbb{N}, \Gamma \vdash g_s\ n' : \mathsf{Gen}_\alpha\ (P\ (n' + 1))}{n{:}\mathbb{N}, \Gamma \vdash \mathsf{match}\ n\ \mathsf{with}\ \mid 0 \Rightarrow g_z \mid n' + 1 \Rightarrow g_s\ n' : \mathsf{Gen}_\alpha\ (P\ n)}\ \textsc{S-SplitNat}$$

Here we've given just the rules for booleans and natural numbers; rules for other inductive types can be derived from their definitions (see Section 5).

3.3 Standard Library Generators

The rules from the previous section are the core of our synthesis algorithm, and they can already be used to build complex generators, but the real power of the approach comes from its extensibility. Rather than ask the synthesis process to synthesize for arbitrary predicates "all the way down," we can provide it with a library of building blocks that it can assemble to build more complex generators. For the examples in the rest of this paper, we will need just a few generators that are standard in PBT libraries.

Choose. The choose generator picks a natural number in a defined range. We define it by recursion

$$\mathsf{def\ choose}\ (lo\ hi : \mathbb{N}) := \mathsf{if}\ lo = hi\ \mathsf{then\ pure}\ lo\ \mathsf{else\ pick}\ (\mathsf{pure}\ lo)\ (\mathsf{choose}\ (lo+1)\ hi)$$

and characterize its support as follows:

Lemma 3.1 (Choose Support)**.** *If $lo \le hi$, then $a \in [\![\mathsf{choose}\ lo\ hi]\!] \iff lo \le a \le hi$.*

Here is the corresponding synthesis rule:

$$\frac{\Gamma \vdash lo \le hi}{\Gamma \vdash \mathsf{choose}\ lo\ hi : \mathsf{Gen}_{\mathbb{N}}\ (\lambda a \Rightarrow lo \le a \le hi)}\ \textsc{S-Choose}$$

In a synthesis context, it may not always be easy to show that $lo \le hi$, or it might not even be true. This motivates a second way to synthesize choose that checks its precondition explicitly.

$$\frac{}{\Gamma \vdash \mathsf{assume}\ lo \le hi\ \mathsf{in\ choose}\ lo\ hi : \mathsf{Gen}_{\mathbb{N}}\ (\lambda v \Rightarrow lo \le v \le hi)}\ \textsc{S-ChoosePartial}$$

Both rules are valid, but the second introduces the potential for the generator to fail: if it is called with $lo > hi$, it will not be able to produce a value. Even so, this generator is sound in the sense that, if it produces a value, then that value satisfies the given condition—but it is not ideal for testing. Luckily, we can usually optimize the assume away later.

Greater Than and Less Than. The greaterThan and lessThan generators take a single natural number and can generate any number respectively greater or less than that number. Their implementation and associated lemmas are similar the ones for choose, so we defer them to Appendix A.

Elements. Finally, the elements generator picks a random value from a list:

$$\mathsf{def\ elements}\ (xs : \mathsf{List}\ \alpha) := \mathsf{match}\ xs\ \mathsf{with}\ \mid [x] \Rightarrow \mathsf{pure}\ x \mid x :: xs' \Rightarrow \mathsf{pick}\ (\mathsf{pure}\ x)\ (\mathsf{elements}\ xs')p$$

It has the following support and synthesis rules:

Lemma 3.2 (Elements Support and Synthesis)**.** *If $xs \ne []$, then $a \in [\![\mathsf{elements}\ xs]\!] \iff a \in xs$.*

$$\frac{\Gamma \vdash xs \ne []}{\Gamma \vdash \mathsf{elements}\ xs : \mathsf{Gen}_{\alpha}\ (\lambda a \Rightarrow a \in xs)}\ \textsc{S-Elements}$$

$$\frac{}{\Gamma \vdash \mathsf{assume}\ xs \ne []\ \mathsf{in\ elements}\ xs : \mathsf{Gen}_{\alpha}\ (\lambda a \Rightarrow a \in xs)}\ \textsc{S-ElementsPartial}$$

3.4 Optimizing Generators to Avoid Assumes

In most cases where the synthesizer inserts assumptions, they can be optimized away. For example, consider the following synthesized generator:

$$\frac{\cdots \qquad \overline{lo{:}\mathbb{N},\, hi{:}\mathbb{N} \vdash \mathsf{assume}\ lo \le hi\ \mathsf{in}\ \mathsf{choose}\ lo\ hi : \mathsf{Gen}\ (\lambda\, a \Rightarrow lo \le a \le hi)}}{lo{:}\mathbb{N},\, hi{:}\mathbb{N} \vdash \mathsf{pick}\ (\mathsf{pure}\ 0)\ (\mathsf{assume}\ lo \le hi\ \mathsf{in}\ \mathsf{choose}\ lo\ hi) : \mathsf{Gen}(\lambda\, a \Rightarrow a = 1 \lor lo \le a \le hi)}$$

Logically, this assume is not necessary. As written, the generator makes a choice and then fails if it happened to choose the right side and $lo > hi$. But it could just as well check $lo \le hi$ first and only choose the right branch if the check succeeds. Generalizing this observation, we design a set of *optimization rules* that rewrite generators to avoid failures. The rule we need for the above case is

$$\mathsf{pick}\ x\ (\mathsf{assume}\ b\ \mathsf{in}\ y) \rightsquigarrow \mathsf{if}\ b\ \mathsf{then}\ \mathsf{pick}\ x\ y\ \mathsf{else}\ x,$$

which rewrites a pick containing an assume to an if statement that checks the assumption before the choice. Concretely, in this case:

$$\mathsf{pick}\ (\mathsf{pure}\ 0)\ (\mathsf{assume}\ lo \le hi\ \mathsf{in}\ \mathsf{choose}\ lo\ hi) \rightsquigarrow$$
$$\mathsf{if}\ lo \le hi\ \ \mathsf{then}\ \mathsf{pick}\ (\mathsf{pure}\ 0)\ (\mathsf{choose}\ lo\ hi)\ \mathsf{else}\ \mathsf{pure}\ 0$$

In total, we use six optimization rules to uncover assumes and lift them out of choices:

$$\begin{array}{rcll}
\mathsf{pure}\ v \ggg f & \rightsquigarrow & f\ v & \qquad (1)\\
(x \ggg g) \ggg f & \rightsquigarrow & x \ggg (\lambda a \Rightarrow g\ a \ggg f) & \qquad (2)\\
(\mathsf{assume}\ b\ \mathsf{in}\ x) \ggg f & \rightsquigarrow & \mathsf{assume}\ b\ \mathsf{in}\ (x \ggg f) & \qquad (3)\\
x \ggg (\lambda\, a \Rightarrow \mathsf{assume}\ b\ \mathsf{in}\ (f a)) & \rightsquigarrow & \mathsf{assume}\ b\ \mathsf{in}\ (x \ggg f) \qquad \text{if } a \notin \mathsf{fv}(b) & \qquad (4)\\
\mathsf{pick}\ (\mathsf{assume}\ b\ \mathsf{in}\ x)\ y & \rightsquigarrow & \mathsf{if}\ b\ \mathsf{then}\ \mathsf{pick}\ x\ y\ \mathsf{else}\ y & \qquad (5)\\
\mathsf{pick}\ x\ (\mathsf{assume}\ b\ \mathsf{in}\ y) & \rightsquigarrow & \mathsf{if}\ b\ \mathsf{then}\ \mathsf{pick}\ x\ y\ \mathsf{else}\ x & \qquad (6)
\end{array}$$

Rules (1) and (2) are standard monad equivalences, (3) and (4) describe how assumes interact with binds, and (5) and (6) actually lift assumes out of choices.

Lemma 3.3 (Optimizations Correct)**.** Rules (1)–(6) do not change the support of the generator.

These rules are not complete; they may still leave assumes in the generator. For example, a pick with assumes on both sides will still fail if both conditions are false. But, for most of the examples in Section 6, they are enough to create a generator that is assumeFree.

4 SYNTHESIZING GENERATORS FOR RECURSIVE STRUCTURES

We next describe how our synthesis procedure handles predicates over inductive data structures like lists and trees. Our approach is based on *recursion schemes*, so we start with some background on those (Section 4.1). Then we outline our synthesis procedure in stages, describing the basic approach (Section 4.2), adapting that approach to work with a wider range of predicates (Section 4.3), adding support for multiple conjoined recursive predicates (Section 4.4), and finally putting everything together (Section 4.5).

At a high level, the approach in this section takes a recursive predicate, normalizes it, and then applies a synthesis rule like the ones from the previous section. The pipeline that we implement is shown in Figure 3 and explained in detail below. The key takeaway is that, rather than give the synthesizer direct access to indexed and recursion, we synthesize all recursive generators through higher-level rules. This approach does have limitations—for example, we cannot directly synthesize generators like elements that iterate over one structure and produce another—but it is highly effective for predicates that directly constrain data structures.

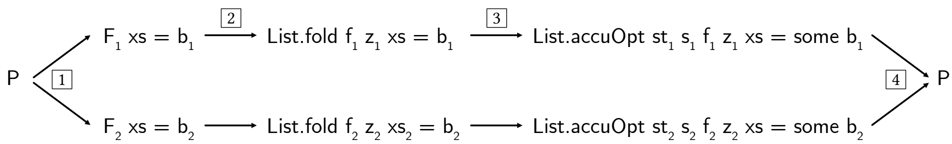

Fig. 3. How a list predicate is normalized for synthesis.

4.1 Background: Recursion Schemes

Recursive functions are a common challenge for program analysis and synthesis tools, even in strongly normalizing languages where they are guaranteed to terminate. While there are techniques for synthesizing recursive programs directly from recursive specifications [43, 44], we adopt a different approach that is simpler and easier to embed in Lean.

The functional programming community has produced a rich literature on *recursion schemes*. Rather than express recursive functions directly via unstructured general recursion, recursion schemes abstract recursion into structured forms that are easier to reason about.

Folds. The simplest recursion scheme is a *fold* or *catamorphism*. Here is an implementation of fold for the List datatype:

$$\begin{array}{l} \text{def List.fold } (f : \alpha \to \beta \to \beta)\ (z : \beta)\ (xs : \text{List } \alpha) : \beta := \\ \quad \text{match } xs \text{ with} \\ \quad \mid [\,] \Rightarrow z \\ \quad \mid x :: xs' \Rightarrow f\ x\ (\text{List.fold } f\ z\ xs') \end{array}$$

This function takes as arguments a “base-case” z and a “step function” f. We call the type β the “collector” for the fold.[3] When the list is empty, we return z. When the list is a cons-cell, we recursively call List.fold f z on its tail and use f to combine the resulting value with the value at the head.

Note that information here flows backward, from the tail of the list to the head.[4] We first compute something about the tail, without considering the value at the head, and only at the end do we actually put the two together. This point will be useful to remember.

More Advanced Recursion Schemes. While List.fold can represent all terminating functions over lists [22], we will see that it is not always ergonomic to do so. For this reason, researchers have identified dozens of specialized recursion schemes, each capturing some common pattern of recursion that arises in functional programs [61]. Of note for this paper is the *accumulation* pattern [41], which we present with some minor simplifications:

$$\begin{array}{l} \text{def List.accu } (st : \alpha \to \sigma \to \sigma)\ (s : \sigma)\ (f : \alpha \to \beta \to \sigma \to \beta)\ (z : \sigma \to \beta)\ (xs : \text{List } \alpha) : \beta := \\ \quad \text{match } xs \text{ with} \\ \quad \mid [\,] \Rightarrow z\ s \\ \quad \mid x :: xs' \Rightarrow f\ x\ (\text{List.accu st } (\text{st } x\ s)\ f\ z\ xs')\ s \end{array}$$

[3]Other texts call this value the “accumulator,” but we use “accumulation” to refer to a type of fold [41] and want to avoid confusion.
[4]I.e., this is a “right fold” over the list (i.e., List.foldr). In this paper we drop the “r” for consistency across data structures: left folds are natural for lists, but they do not have an analog for algebraic data types with branching recursion like trees.

Accumulations are similar to folds, but they pass information in both directions. The collector value β is still passed backwards through the list, but we add a new type parameter σ representing an "accumulation state" that flows forward. The accumulation takes an initial state s as input, along with a "state update function" st that says how the state changes based on the head of the list.

We can use List.accu to naturally implement some functions that would be awkward with List.fold. For example, this function checks if a list of natural numbers is sorted:

$$\text{def sorted } xs := \text{List.accu } (\lambda\, x\ \_ \Rightarrow x)\ 0\ (\lambda\, x\ b\ lo \Rightarrow lo \le x \wedge b)\ (\lambda\, \_ \Rightarrow \text{true})\ xs$$

The accumulation state is the minimum value allowed in the remainder of the list; it is initialized to 0 and updated to the most recently seen value at each step. The step function then checks that $lo \le x$ and conjoins that with the boolean computed from the tail of the list, to ensure that the tail of the list is also sorted. We could implement this same function with List.fold, but this would require the collector to be a higher-order function. Higher-order functions can be difficult to work with so accumulations provide a convenient alternative representation.

Optional Folds. We also need versions of List.fold and List.accu that capture *optional computations*:

```
def List.foldOpt
    (f : α → β → Option β)
    (z : Option β)
    (xs : List α) : Option β :=
  match xs with
  | [] ⇒ z
  | x :: xs′ ⇒
    match List.foldOpt f z xs′ with
    | none ⇒ none
    | some b′ ⇒ f x b′
```

```
def List.accuOpt
    (st : α → σ → σ) (s : σ)
    (f : α → β → σ → Option β)
    (z : σ → Option β)
    (xs : List α) : Option β :=
  match xs with
  | [] ⇒ z s
  | x :: xs′ ⇒
    match List.accuOpt st (st x s) f z xs′ with
    | none ⇒ none
    | some b′ ⇒ f x b′ s
```

We discuss use cases for these in the next section; for now, notice that they behave the same as List.fold and List.accu, except that, if any step evaluates to none, then the whole thing does.

Unfolds. The final recursion scheme we will examine in detail is the *unfold* or *anamorphism*. Unfolds are the inverse of folds: whereas folds collapse data structures into compact values, unfolds expand values into data structures. Here is an example:

```
def List.unfold (g : β → Gen (Option (α × β))) (b : β) : Gen (List α) :=
  let rec go b fuel :=
    match fuel with
    | 0 ⇒ pure none
    | 1 + fuel′ ⇒
      g b >>= λ step ⇒
      match step with
      | none ⇒ pure (some [])
      | some (x, b′) ⇒
        go b′ fuel′ >>= λ mxs ⇒
        match mxs with | none ⇒ pure none | some xs ⇒ pure (some (x :: xs))
  indexed (go b)
```

The internal function go takes a seed value b and some fuel. If the fuel is gone, it returns none. Otherwise, it samples $g\ b$ to obtain a "step"—if the step is none then generation terminates with an empty list, and, if the step is some (x, b'), then generation continues with a node containing the value x and a new seed value b'. Finally, the indexed constructor unifies this family of generators into a single generator for lists.

A key benefit of List.unfold is that it is guaranteed to make exactly one recursive call for each element of the list it produces. This is important for efficiency: it means that generators implemented with List.unfold (as opposed to arbitrary general recursion) are guaranteed to be efficient as long as their step functions are efficient.

4.2 Generators for Inductive Data Types

We now show how the powerful tools for structuring recursion that we reviewed in the previous subsection allow our synthesis procedure to handle predicates over inductive data structures.

For a first example, consider the following predicate, which uses List.fold to check that a list has a given length:

$$\text{def isLengthK } (k : \mathbb{N})\ (xs : \text{List Nat}) := (\text{List.fold } (\lambda\, x\, b \Rightarrow 1 + b)\ 0\ xs) = k$$

We can also use List.unfold to write a generator for values satisfying this predicate:

$$\begin{array}{l}
\text{def genLengthK } (k : \mathbb{N}) : \text{Gen } (\text{List } \mathbb{N}) := \\
\quad \text{List.unfold} \\
\quad\quad (\lambda\, n \Rightarrow \\
\quad\quad\quad \text{match } k \text{ with} \\
\quad\quad\quad \mid 0 \Rightarrow \text{pure none} \\
\quad\quad\quad \mid 1 + k' \Rightarrow \text{arbNat} \gg\!= \lambda\, x \Rightarrow \text{pure } (\text{some } (x, k'))) \\
\quad\quad n
\end{array}$$

At each unfolding step, the generator checks the seed value n. If $n = 0$ then it generates none, indicating that the list should end (since n is the target length of the list). Otherwise, it generates an arbitrary natural number x and yields some $(x, n - 1)$ to indicate that the list should continue with a cons-cell containing x, plus a new target length.

How might we derive genLengthK from issLengthK? The key observation is that issLengthK and genLengthK have an inverse relationship—whenever genLengthK takes a step, it is guaranteed that isLengthK can undo that step. We can make this observation concrete with the following propositions:

$$\begin{array}{l}
\text{none} \in [\![(\lambda\, n \Rightarrow \\
\quad \text{match } k \text{ with} \\
\quad \mid 0 \Rightarrow \text{pure none} \\
\quad \mid 1 + k' \Rightarrow \\
\quad\quad \text{arbNat} \gg\!= \lambda\, x \Rightarrow \\
\quad\quad \text{pure } (\text{some } (x, k'))) \ b]\!] \\
\iff \\
b = 0
\end{array}
\qquad
\begin{array}{l}
\text{some } (x, b') \in [\![(\lambda\, n \Rightarrow \\
\quad \text{match } k \text{ with} \\
\quad \mid 0 \Rightarrow \text{pure none} \\
\quad \mid 1 + k' \Rightarrow \\
\quad\quad \text{arbNat} \gg\!= \lambda\, x \Rightarrow \\
\quad\quad \text{pure } (\text{some } (x, k'))) \ b]\!] \\
\iff \\
b = (\lambda\, x\, b \Rightarrow 1 + b)\ x\ b'
\end{array}$$

We can see that genLengthK's step function returns none precisely when b is 0—the initial collector value for isLengthK. Likewise, it returns some (x, b') precisely when b is $1 + b'$—which is the result of applying isLengthK's step function to x and b'. More generally:

Lemma 4.1 (Fold-Unfold-Inverse for Lists)**.** If, for all values b, the following relationship holds between an unfold's step function g and a fold's arguments f and z,

$$\begin{aligned} \mathsf{none} \in [\![g\ b]\!] &\iff b = z \\ \forall\ x\ b',\ \mathsf{some}\ (x, b') \in [\![g\ b]\!] &\iff b = f\ x\ b', \end{aligned}$$

then the following holds of the unfold and fold:

$$\forall\ xs,\ xs \in [\![\mathsf{List.unfold}\ g\ b]\!] \iff \mathsf{List.fold}\ f\ z\ xs = b$$

The informal argument for this lemma's correctness bears repeating: the fold and unfold are inverses because, for each step the unfold takes, the fold is guaranteed to be able to "fold that step back up" and recover the seed. Another perspective comes from the observation that a fold passes information backwards in a list from the tail to the head; the unfold does the opposite, passing information forwards in such a way that the fold would always compute the same information going the other direction.

We can use Lemma 4.1 to prove the following synthesis rule correct:

$$\frac{\Gamma \vdash g : (b : \beta) \to \mathsf{Gen}_{\mathsf{Option}\ (\alpha\times\beta)}\ (P\ b)}{\Gamma \vdash \mathsf{List.unfold}\ g\ b : \mathsf{Gen}_{\mathsf{List}\ \alpha}\ (\lambda\ xs\ \Rightarrow \mathsf{List.fold}\ f\ z\ xs = b)}\ \textsc{S-List-Unfold}'$$
$$\text{where}\quad P\ b = \lambda\ step \Rightarrow (step = \mathsf{none} \land z = b) \lor (\exists\ x\ b',\ step = \mathsf{some}\ (x, b') \land f\ x\ b' = b)$$

Indeed, our system can use this rule to synthesize genLengthK from the definition of isLengthK.

4.3 Handling More Complex Folds

The S-List-Unfold' rule works for predicates whose folds have exact matches as unfolds, but other folds require preprocessing to ensure synthesis is effective. For example, consider a simple predicate that checks that all elements of a list are equal to 2:

$$\mathsf{def\ allTwo}\ (xs : \mathsf{List\ Nat}) := \mathsf{List.fold}\ (\lambda\ x\ b \Rightarrow x = 2 \land b)\ \mathsf{true}\ xs = \mathsf{true}$$

Using S-List-Unfold', we could turn this directly into a generator of the form:

```
List.unfold
  (λ b ⇒
    match b with
    | true ⇒ pick (pure none) (pure (some (2, true)))
    | false ⇒ · · · ) true,
```

But note that the false branch will never be executed: b starts as true and remains true every step through the unfold. We would prefer to avoid synthesizing the false branch at all.

The key observation is that allTwo has a hidden invariant that the S-List-Unfold' cannot make use of—if the step function ever returns false, the whole fold returns false. We can make this invariant available to the synthesizer by reinterpreting allTwo as an optional fold:

```
def allTwo (xs : List Nat) :=
  List.foldOpt (λ x () ⇒ if x = 2 then some () else none) (some ()) xs = some ()
```

Now β is Unit, and the step function simply checks if $x = 2$ and, if not, fails. The invariant we wanted falls out of the definition of List.foldOpt—if the step function fails at any step, the whole computation fails.

All we need now is a synthesis rule for List.foldOpt:

$$\frac{\Gamma \vdash g : (b : \beta) \longrightarrow \mathsf{Gen}_{\mathsf{Option}\ (\alpha\times\beta)}\ (P\ b)}{\Gamma \vdash \mathsf{List.unfold}\ g\ b : \mathsf{Gen}_{\mathsf{List}\ \alpha}\ (\lambda\ xs\ \Rightarrow \mathsf{List.foldOpt}\ f\ z\ xs = \mathsf{some}\ b)}\ \textsc{S-List-Unfold}''$$

where $P\ b =$

$$\lambda\ step \Rightarrow (step = \mathsf{none} \land z = \mathsf{some}\ b)\ \lor (\exists\ x\ b',\ step = \mathsf{some}\ (x, b') \land f\ x\ b' = \mathsf{some}\ b)$$

This rule looks roughly the same as S-List-Unfold', but it enforces the step-wise invariant we are interested in. Now, if we know the fold should return some, we can also assume each step should return some. We can use the new rule to obtain the following much simpler generator:

```
def genAllTwo : Gen (List Nat) :=
  List.unfold (λ () ⇒ pick (pure none) (pure (some (2, ())))) ()
```

To reach our most general recursion synthesis rule, we target List.accuOpt rather than List.foldOpt. We define one more synthesis rule, S-List-Unfold, that subsumes and replaces both S-List-Unfold' and S-List-Unfold'':

$$\frac{\Gamma \vdash g : (b : \beta) \longrightarrow (s : \sigma) \longrightarrow \mathsf{Gen}_{\mathsf{Option}\ (\alpha\times(\sigma\times\beta))}\ (P\ b\ s)}{\Gamma \vdash \mathsf{List.unfold}\ g'\ (b, s) : \mathsf{Gen}_{\mathsf{List}\ \alpha}\ (\lambda\ xs\ \Rightarrow \mathsf{List.accuOpt}\ st\ s\ f\ z\ xs = \mathsf{some}\ b)}\ \textsc{S-List-Unfold}$$

where $P\ b\ s =$

$$\lambda\ step \Rightarrow (step = \mathsf{none} \land z\ s = \mathsf{some}\ b)\ \lor (\exists\ x\ b',\ step = \mathsf{some}\ (x, b') \land f\ x\ b'\ s = \mathsf{some}\ b)$$

```
g' b s =
  g b s >>= λ mstep ⇒
  match mstep with
  | none ⇒ pure none
  | some (x, b') ⇒ pure (some (x, (b', st x s)))
```

This allows for both a collector that passes information backward from the tail of the list and a state that flows forward, allowing for the desired short-circuiting behavior.

If we rewrite sorted from the beginning of this section as

```
def sorted xs :=
  List.accuOpt (λ x _ ⇒ x) 0 (λ x b lo ⇒ if lo ≤ x then some () else none) (some ()) xs,
```

we can use S-List-Unfold to derive:

```
def genSorted : Gen (List Nat) :=
  List.unfold
    (λ (lo, ()) ⇒
      pick (pure none) (pick (greaterThan lo) (pure lo) >>= λ x ⇒ pure (some (x, (x, ())))))
    (0, ())
```

This generator behaves the same as the one an expert user might write. At each step, it either ends the list or generates a new value x that is greater than or equal to lo, puts that value in the list, and continues with $lo = x$.

4.4 Tupling Predicates

The S-List-Unfold rule works for predicates that are written as a single pass over a data structure, but sometimes predicates have multiple independent constraints. For example,

$$\mathsf{def\ allTwoLengthK}\ (k : \mathbb{N})\ (xs : \mathsf{List\ Nat}) := \mathsf{allTwo}\ xs \land \mathsf{isLengthK}\ k\ xs$$

combines two predicates that we have seen before into a single predicate.

We have two options for handling such situations. The first is to draw from the literature on program calculation and apply a *tupling* transformation [6, 42] to combine the conjuncts. These transformations were originally designed for optimizing functional programs, but they are a natural fit for this problem. Another is to adapt the "merging" procedure for inductive relations described by Prinz and Lampropoulos [45]. It turns out that, in the case of predicates written with List.accuOpt, these concepts coincide!

We introduce a transformation tupleAccuOpt which takes two predicates P and Q, each expressed with accuOpt, and combines them into a single predicate (also expressed with accuOpt) that computes $P\ xs \land Q\ xs$. The definition is surprisingly straightforward—the state and collector arguments are simply combined in a tuple and computed in parallel—so we do not replicate it here. This transformation means that our synthesis approach automatically benefits from the merging optimizations that users can apply manually in QuickChick [45].

Tupling is not the only useful program transformation proposed for recursive programs in the program calculation literature. If we find that other transformations (e.g., fusion) are useful for the kinds of predicates that users care about, we could easily extend our pipeline with them.

4.5 Putting it All Together

The machinery described in the previous subsections is assembled into a standardized workflow for handling predicates on inductive data as follows (see Figure 3).

(1) Normalize the predicate to be of the form: $\lambda\ xs \Rightarrow F_1\ xs = b_1 \land \cdots \land \Rightarrow F_n\ xs = b_n$.
(2) Rewrite each F as a fold. Concretely, search for z and f satisfying the equations $F\ [] = z$ and $F\ (x :: xs) = f\ x\ (F\ xs)$. If these equations are satisfied, then F can be rewritten as List.fold $f\ z$; this is sometimes referred to as the "universal property of fold" [22, 32].
(3) Rewrite each fold as an optional accumulation based on its return type. For example, allTwo, which collects an always-true boolean, would take a different path from sorted which collects a higher-order function.
(4) Tuple the n different branches of the predicate together.

After normalization, we apply S-List-Unfold to the final accumulation and obtain a generator by recursively synthesizing the step function. This workflow is automated through tactics in Lean, which we discuss in the next section.

Everything in this section has been phrased in terms of the List data type, but there is actually nothing in the workflow that is specific to lists. Any recursive data type that are composed of products and sums admits operations analogous to List.fold, List.accuOpt, etc.[5] This means that the pipeline from Figure 3 directly generalizes to a wide range of recursive data structures.

The case studies in Section 6 required implementing this pipeline (along with other utilities, e.g., for case splitting) for five data structures: lists, binary trees, STLC types, STLC terms, and stacks (from [21]). The implementation process is quite mechanical, as the definitions follow the structure of the data type and its constructors, and we believe it will be straightforward to automate it, either via meta-programming or by leveraging *quotients of polynomial functors* (QPFs) [5]. We leave this engineering exercise as future work.

5 PALAMEDES: SYNTHESIZING GENERATORS IN LEAN

In this section we describe Palamedes, our implementation of PBT generator synthesis. We begin with an overview (Section 5.1), then describe the synthesis algorithm in detail (Section 5.2).

[5]Technically, such functions exist for all algebraic data types that arise as the least fixed point of a traversable functor [33].

5.1 Overview

Palamedes is packaged as a library for the Lean theorem prover. Because of Lean's powerful meta-programming capabilities, everything can be implemented as standard Lean code, and using it requires only importing the library into client code.

Starting with a definition of the BST validity predicate

```
def isBST (t : Tree Nat) : Nat × Nat -> Bool := fun (lo, hi) =>
  match t with
  | leaf => true
  | node l x r =>
      (lo <= x && x <= hi) && isBST (lo, x - 1) l && isBST (x + 1, hi) r,
```

we can use Palamedes to synthesize a generator of valid BSTs:

```
def genBST (lo hi : Nat) : Gen (Tree Nat) := by
  generator_search (fun t => isBST t (lo, hi) = true)
```

In general, the user defines some predicate that they want their data to satisfy (such as isBST above), and they use the `generator_search` tactic to find an efficient generator that is sound and complete with respect to that predicate. The above call finds the following generator:[6]

```
Tree.unfold
  (fun ((), lo, hi) =>
    if lo <= hi then
      pick (pure none)
            (choose lo hi (...) >>= fun x =>
            pure (some (((), lo, x - 1), x, ((), x + 1, hi))))
    else pure none) ((), lo, hi)
```

This code uses `Tree.unfold` to manage recursion and termination, but it operates exactly the same way as `genBST` from Section 2. The synthesized code can be pasted directly into the user's file; from there, they can modify it as they see fit (e.g., to tweak the distribution).

While there are no proofs visible to the user in this workflow, they exist under the hood. The `generator_search` tactic proves that the generator it synthesizes is sound and complete with respect to the provided predicate and also assume-free. Palamedes provides alternative versions of `generator_search` that give users access to those proofs if needed.

5.2 The Synthesis Algorithm

At a high level, "`generator_search P`" implements the following steps (expressed as a series of Lean tactic statements):

```
let g  : CorrectGen P  := by synthesize
let g' : CorrectGen P  := by optimize g
let _  : AssumeFree g' := by prove_assume_free
exact g'
```

First, we synthesize an initial generator `g`. This generator has type `CorrectGen P` (it was written $\mathrm{Gen}_\alpha\ P$ earlier). It is implemented in Lean as:

```
def CorrectGen {α : Type} (P : α -> Prop) := {g : Gen α // ∀ v, v ∈ ⟦g⟧ <-> P v}
```

(In Lean this type is called a "subtype"; it is a dependent pair of a value and a proof that it satisfies a given predicate. A value of type `CorrectGen P` is therefore a pair of a generator and a proof that its support is equivalent to `P`.) Next, we optimize the generator by applying the rewrites described

[6] For readability, we manually added a pattern match on `((), lo, hi)`—the actual synthesized code uses projections.

in Section 3.4; this procedure also produces a `CorrectGen`, ensuring that optimization has not changed the support of the generator. Finally, we try to prove that the generator is assume-free (Definition 3.3). If the generator is not assume-free (e.g., in the AVL tree example in Section 6), we emit a warning for the user. We now describe these steps in more detail.

Step 1: Synthesize. The `synthesize` tactic solves a goal of type `CorrectGen P` by applying the synthesis rules described in Section 3.2. The procedure uses Aesop, a tactic in Lean that performs best-first proof search [30]. Aesop takes a large list of Lean tactics and applies them in a loop; when all tactics fail to solve a particular sub-goal (e.g., when the search tries generating one existentially-defined variable first, but it should have tried another), the search backtracks to try a different route. Relying on Aesop in this way turned out to be extremely effective both in terms of the results for our case studies and in terms of ease of implementation.

The rules we provide to Aesop mirror the ones described in Section 3; they are given in Table 1. Each core rule applies a function that builds a term of type `CorrectGen P` for some `P`; for example, S-Pick is defined as:

```
def s_pick (x : CorrectGen P) (y : CorrectGen Q)
          : CorrectGen (fun a => P a ∨ Q a) :=
  Subtype.mk (pick x y) (...)
```

This combines the rule for synthesizing `pick` with a proof (elided in the above definition) that the rule is valid with respect to the definition of the generator's support . Every synthesis rule is constructed this way, ensuring that the end-to-end synthesis process is correct by construction.

As discussed in Section 3.2, the synthesis rules may not apply to a given goal directly. The Convert rule allows the synthesizer to change the predicate to an equivalent one, as long as we can prove the equivalence:

```
def convert (h : Q = P) (x : CorrectGen P) : CorrectGen Q := Subtype.mk x (...)
```

In practice, rather than allowing the synthesizer to apply `convert` arbitrarily, we combine it with the various generator builders. For example, the synthesizer can use

```
apply convert (by match_pick) (s_pick _ _),
```

to apply `convert` with a pre-determined end goal (the conclusion of `s_pick`). To see this in action, consider the goal:

```
CorrectGen (fun a => (a = 3 ∨ a = 2) ∧ True)
```

Simply trying to `apply s_pick` here will fail, but if the synthesizer applies the `convert` version then `match_pick` can simplify away the no-op term and apply the rule.

As mentioned in Section 3, the rule for `s_bind` is especially dependent on `convert`. At a given point in the synthesis process, `s_bind` may apply in a number of different ways, corresponding to different orders that values may be generated in. Concretely, for the predicate $\exists\ x\ y, \ldots$, it is not clear whether to apply the bind rule to x or y first. For this reason, we need multiple versions of the `s_bind` rule, each targeting a different generation order. For the examples in the following section, we only need two: the version using `match_bind 0` tries prioritizing the first existentially quantified variable and `match_bind 1` tries the second. More complex examples may require more of these rules. The `match_bind` tactic also rearranges conjuncts in the predicate to better match the pattern required by `s_bind`.

For inductive data types, we follow the same pattern as the core rules. Rules like

```
apply convert (by match_List_unfold) (List.s_unfold _)
```

implement the pipeline from Figure 3, turning user-written recursive predicates into calls to `List.accuOpt`.

Table 1. The synthesis rules used to synthesize our core examples. Each rule has a precedence; rules with 100% precedence are tried first and never backtracked. Other rules are tried in order of precedence, and higher-precedence branches of the proof are explored first. Rules containing <T> are replicated once for each of the recursive data structures we consider.

Rule	Precedence
uncurry_intro	100%
assumption	100%
apply convert (by match_pure) s_pure	100%
apply convert (by match_pick) (s_pick _ _)	50%
apply convert (by match_bind 0) (s_bind _ _)	50%
apply convert (by match_bind 1) (s_bind _ _)	50%
apply convert (by match_greaterThan) s_greaterThan	50%
apply convert (by match_lessThan) (s_lessThan (by solve_lessThan))	50%
apply convert (by match_lessThan) s_lessThan_partial	50%
apply convert (by match_between) (s_between (by solve_between))	50%
apply convert (by match_between) s_between_partial	50%
apply convert (by match_elements) (s_elements (by solve_elements))	50%
apply convert (by match_elements) s_elements_partial	50%
apply convert (by match_<T>_unfold) (<T>.s_unfold _)	50%
split_cases 0 <T>.split	5%
split_cases 1 <T>.split	5%

Finally, we implement the S-Split* rules similarly to `s_bind`. For example, the tactic

```
split_cases 1 Nat.split
```

looks for the second (0-indexed) `Nat` in the context and attempts to split it with S-SplitNat.

Step 2: Optimize. The `optimize` tactic applies the optimization rules presented in Section 3.4. It is implemented as a meta-level function, operating directly on the AST of the generator. This makes it easy to write rules like rule (4) from Section 3.4, which matches on the body of a lambda abstraction. Being written at the meta level means that we cannot prove once and for all that optimization is correct in Lean (though it is straightforward to prove on paper). Instead, we use proof automation to show that each optimized generator is equivalent to its unoptimized counterpart.

Step 3: (Optionally) Prove Assume-Free. While Palamedes does not categorically reject generators that contain assumes, since they can still be useful in some cases, it does use straightforward proof automation to check that a generator does not make nontrivial use of the assume constructor. If it fails, it outputs a warning.

Step 4: Render to the User. At this point, the user has a choice. If they think their predicate may change over the course of development, or if they simply want to simplify the codebase, they can choose to leave the call to `generator_search` as the definition of their generator. Lean will try to cache the generator when possible, and otherwise it will re-synthesize the generator when reloading the file. Users may also want to *render* the synthesized generator as a concrete program. They might, for example, want to add weights to bias the distribution or add size bounds to ensure generated values do not get too big.[7] When the user wants to render a generator, they can use a variant of the search tactic (`generator_search?`). This version invokes the same synthesis procedure and then

[7]This may make the generator incomplete, so we do not do it during synthesis, but an expert user may decide the incompleteness is worth it on a case-by-case basis.

provides the user with a “try this” widget [8] in their editor. Clicking on the hint pastes the full text of the generator into their file, and they can then edit as normal.

We make a couple of choices during the synthesis process to make rendering possible, both of which address subtle technical details. First, we mark all `CorrectGen` constructors as `reducible`. This tells Lean’s evaluator that those definitions can (and should) be reduced during elaboration. Second, we ensure that all `CorrectGen` constructors are of the form `Subtype.mk g pf`, where `g` is a plain generator and `pf` is a proof about that generator. This means that, when we project out the generator `g`, it is guaranteed to be independent of the proof. In some other type theories [2], all terms of type `CorrectGen` could be guaranteed to reduce to a call to `Subtype.mk`; but in Lean, a term might reduce to a type cast (i.e., `h ▸ e`) from which we cannot readily extract the generator component. These choices do slightly complicate our synthesis procedure—it means we need to use Aesop somewhat cautiously—so we hope to be able to relax these requirements in the future.

Step 5: Interpret and Run the Generator. The final step of the process is to actually test code with the generator by sampling from it. As discussed in Section 3, the generators we synthesize are data structures, not programs, so they cannot be run directly. The *sampling interpreter* gives meaning to generators as maps from random seeds to values, mirroring Definition 3.1. In order to be consistent with a generator’s support, the sampling interpretation needs to handle backtracking and non-termination carefully. Specifically, it re-samples values in the case of failure (e.g., from assume), and functions wrapped by indexed are given exponentially increasing fuel if they run out.

6 EVALUATION

In this section, we evaluate our synthesis algorithm by using Palamedes to synthesize a set of benchmark generators. Although our synthesis procedure is not complete (see Section 7), this demonstrates that it works for a wide range of interesting and useful predicates. Concretely, we address two key research questions:

RQ1 Can Palamedes synthesize a variety of generators in an acceptable time frame?
RQ2 Are the generators that Palamedes synthesizes comparable to ones that expert users write?

We consider the first of these questions in Section 6.1 and the second in Section 6.2.

6.1 Benchmark Overview and Timing

To answer **RQ1**, we benchmark Palamedes on a variety of predicates. Some demonstrate specific aspects of the synthesis algorithm (including low-level examples that appear in the text above), while others are drawn from the literature on PBT [26, 28, 45].

A selection of the benchmark predicates is presented in Table 2; the rest (43 total) appear in Appendix C. The synthesis times range from around 40 milliseconds for very basic examples to up to 34 seconds for a complex example with multiple nested case splits (AVL trees). It takes under 2 seconds to synthesize `genBST` and under 4 seconds to synthesize a generator for well-typed STLC terms.

We contend that these synthesis times are well within the acceptable range for our intended deployment setting. Our assumption, supported by prior work [16], is that PBT developers tend to run tests far more frequently than they change definitions. Thus, the cost of synthesis will be amortized across many test runs. (By contrast, the costs incurred by search-based approaches to generation are paid every time the developer runs their tests.)

Backtracking Generators. The vast majority of our synthesized generators are assume-free; only three—two versions of AVL trees and a demonstration example—have assumes.

[8] https://lean-lang.org/documentation/widgets/

Table 2. Benchmark predicates and synthesis times. The value being generated is **v**; other variables are universally quantified unless specified. External definitions (e.g., `isBST`) are presented in Appendix B. Generators above the line are assume-free; the ones below are not. Benchmarks were run on an M1 MacBook Pro with 8 cores and 16 GB of memory using Lean v4.21.0. Times are averaged over 30 runs; means are presented with standard deviations in parentheses. All times are in seconds.

Predicate	Type	Time (s)	
`v = 2`	`Nat`	0.04	(0.01)
`2 = v`	`Nat`	0.04	(0.00)
`v = 2 ∨ v = 5`	`Nat`	0.08	(0.00)
`v = 2 ∨ v = 5 ∧ True`	`Nat`	0.08	(0.00)
`∃ a, a = 3 ∧ v = a + 1`	`Nat`	0.04	(0.00)
`5 <= v ∧ v <= 10`	`Nat`	0.08	(0.00)
`v > 5`	`Nat`	0.07	(0.00)
`v = 0 ∨ lo <= v ∧ v <= hi`	`Nat`	0.14	(0.00)
`isAllTwos v = true`	`List Nat`	0.84	(0.01)
`isAllTwosEvenLen v = true`	`List Nat`	2.76	(0.02)
`isEvenLen v = true`	`List Nat`	2.22	(0.02)
`isIncreasingByOne v = true`	`List Nat`	1.44	(0.01)
`List.length v = k`	`List Nat`	1.89	(0.01)
`isLengthKAllTwos k v = true`	`List Nat`	2.37	(0.01)
`isSortedBetween v (lo, hi) = true`	`List Nat`	1.72	(0.02)
`isTrue v = true`	`List Nat`	2.21	(0.01)
`isAllTwos v = true`	`Tree Nat`	0.61	(0.01)
`isBST v (lo, hi) = true`	`Tree Nat`	1.86	(0.01)
`isComplete v n = true`	`Tree Nat`	2.35	(0.02)
`isIncreasingByOne v = true`	`Tree Nat`	1.57	(0.02)
`isNonempty v = true`	`Tree Nat`	1.66	(0.04)
`isGoodStack v n = true`	`Stack`	5.39	(0.05)
`isWellScoped v 0 = true`	`Term`	2.85	(0.03)
`isWellTyped Γ v`	`Term`	3.66	(0.03)
`lo <= v ∧ v <= hi`	`Nat`	0.07	(0.00)
`isAVL height lo hi v = true`	`Tree Nat`	34.44	(0.22)

Generating AVL trees without backtracking is tricky. A common failure mode, even for many hand-written AVL tree generators, is running out of valid values to use as node labels before some branch is deep enough to be balanced. Indeed, the AVL tree generator that Prinz and Lampropoulos [45] present in their paper on merging inductive relations also backtracks in this case. However, there are ways to write AVL tree generators that do not backtrack—we show two in Appendix F. The first version more carefully selects values for each node. Rather than choose a value between `lo` and `hi`, it leaves `2 ^ (height - 1)` values on either side of the range to ensure that the next step of generation will not run out of values. One might hope Palamedes would be able to find such a generator, but this would require automatically rewriting the predicate far beyond what is currently possible. The other option generates AVL trees by repeatedly inserting values into an empty tree. We would never expect Palamedes to find this generator—it would require far too much domain knowledge—but it is probably the approach an experienced PBT user would reach for in practice.

```
def genWellTyped (Γ : List Ty) : Gen Term := by
  let τ <- arbTy; Term.unfold
    (fun (τ, Γ) => do
      let step <- caseTy τ
        (fun () =>                                                  -- τ = .unit
          pick
            (pure TermF.unitStep)
            (if (Γ.indexesOf .unit).length > 0 then
              pick (do let n <- elements (Γ.indexesOf .unit) (...)
                    pure (TermF.varStep n))
                   (do let τ' <- arbTy
                       pure (TermF.appStep (.arrow τ' .unit) τ'))
            else do
              let τ' <- arbTy; pure (TermF.appStep (.arrow τ' .unit) τ')))
        (fun τ1 τ2 ()   =>                                          -- τ = .arrow τ1 τ2
          if (Γ.indexesOf (.arrow τ1 τ2)).length > 0 then
            pick (do let n <- elements (Γ.indexesOf (.arrow τ1 τ2)) (...)
                  pure (TermF.varStep n))
                 (pick
                   (pure (TermF.absStep τ1 τ2))
                   (do let τ' <- arbTy
                       pure (TermF.appStep (.arrow τ' (.arrow τ1 τ2)) τ')))
          else
            pick (pure (TermF.absStep τ1 τ2))
                 (do let τ' <- arbTy
                     pure (TermF.appStep (.arrow τ' (.arrow τ1 τ2)) τ')))
      match step with
      | TermF.unitStep => pure TermF.unitStep
      | TermF.varStep n => pure (TermF.varStep n)
      | TermF.absStep τ b => pure (TermF.absStep τ (b, τ :: Γ))
      | TermF.appStep b1 b2 => pure (TermF.appStep (b1, Γ) (b2, Γ))) (τ, Γ)
```

Fig. 4. Synthesized generator for well-typed STLC terms. Comments were added manually, and some variables were renamed for clarity.

6.2 Synthesized vs. Handwritten Generators

Now we turn to **RQ2**, exploring how the generators that Palamedes produces compare to ones that expert users might write.

As our main case study, we focus on the generator for well-typed STLC terms that Palamedes synthesizes, shown in Figure 4. Following the S-Bind rule, `genWellTyped` picks a random type and then generates a random term of that type—is the method that prior work [38] suggests for a generator like this. The bulk of the generator uses `Term.unfold` to generate terms step by step. Each step, the generator inspects the type and produces a `TermF`, which determines the next data constructor that should be generated. (`TermF` is analogous to the `Option` $(\alpha \times \beta)$ that appears in the type of `List.unfold`.) In both cases (unit or arrow), the generator can choose to select a variable from the context, generate a function application, or generate a type-specific term—the unit value or a lambda abstraction. Once the next step is chosen, the type of the continuation is packaged up with a modified context and used to seed the next step.

This generator is logically the same as one that an expert user might write to satisfy the same predicate, although the code would be a bit different in its details. An expert user would reduce repetition by waiting to split on τ until it is necessary to do so. They would likely also avoid the match statement at the end, which is an artifact of the synthesis process. Still, the asymptotic performance of the generators would be the same—neither would backtrack, and both would contain the same basic clauses. For comparison, we provide a hand-written STLC generator that the authors wrote in Appendix D.

We can also compare the synthesized generator to ones described in the literature, in particular the STLC generators from the Etna PBT evaluation framework [50]. (These generators are available online, but we replicate the Haskell and Rocq versions of the generator in Appendix E for easy reference.) Focusing on the Haskell version first, this has some code to carefully control the sizes of generated values that the synthesized version does not (we discuss this limitation of PALAMEDES in Section 7). But the high-level control flow of the Haskell version `genExpr` is the same as PALAMEDES's `genWellTyped`. Interestingly, the QuickChick version of the STLC generator in the Etna suite, which is automatically derived from an inductive relation specifying well-typedness [28], is *not* logically equivalent to the other versions. In the case where the context does not contain a variable of the appropriate type, the QuickChick version may try to generate a bound variable and then have to backtrack. This is not terribly wasteful, but it is sub-optimal.

Appendix F shows five more generators that PALAMEDES synthesized, side-by-side with ones that we wrote manually. For all but the last one (AVL trees, which we discuss at length above), there were no functional differences between the generators we wrote and the synthesized ones.

To summarize, when PALAMEDES finds an assume-free generator for a predicate, it is faithful to the kinds of generators that expert users write for those same predicates.

7 LIMITATIONS

Our approach also has some limitations that are important to understand.

First, our synthesis algorithm is necessarily partial; there are many Lean predicates that PALAMEDES cannot handle. For example, PALAMEDES's tupling and fold normalization pipeline is not currently powerful enough to synthesize a generator for red-black trees [28]. Additionally, since all uses of assume need to be explicitly included in the generator library (e.g., in the definitions of S-CHOOSE and S-ELEMENTS), new generators that need assumptions cannot be automatically synthesized. Luckily, PALAMEDES is extremely extensible—users can add new proof automation, library generators, and synthesis rules—so its partiality can be mitigated over time. In Section 9, we discuss plans to extend the set of predicates that PALAMEDES can handle.

One classic PBT generator feature that PALAMEDES does not yet incorporate is *sizes*. QuickCheck generators [11] have built-in ways to vary the size of generated values over the course of generation, which we do not include in our Gen type. Sizes are compatible with our approach—prior work [40] gives a semantics for internal sizing that plays well with our definitions—but we do not yet have a heuristic for how the synthesizer should use them. Currently, there are two ways to control the sizes of generated values in PALAMEDES. First, users can augment their specifications with explicit size constraints. For example, `isLengthKAllTwos` allows the user to control the length of generated lists by changing the argument `k`; the tupling transformation means this can be done by simply conjoining an extra predicate. Alternatively, users can manually tweak their generators after synthesis to add size control.

8 RELATED WORK

We discuss related work on constrained random generation, on generator synthesis, and on deductive synthesis more generally.

The Constrained Random Generation Problem. As discussed at length in Section 2.3, many approaches to the constrained random generation problem have been proposed over the years [10, 17, 47, 49, 51, 57]. These all solve a slightly different problem from the one solved by PALAMEDES: they actively guide generation towards valid values *during testing*, rather than searching for a sound and complete generator ahead of time. In situations where a developer writes their generator once and then runs it many times—common in PBT [16]—these approaches may not scale as well as ones that synthesize generators.

The Constrained Generator Synthesis Problem. While most existing work opts to search for inputs during generation, a few papers do tackle the constrained generator synthesis problem, more or less as we present it.

In the Rocq theorem prover [52], QuickChick provides an automated mechanism for deriving generators that are sound and complete with respect to predicates defined as inductive relations [28, 39]. This mechanism is extremely effective at efficiently deriving high quality generators, but there are some clear trade-offs between QuickChick's approach and ours. On one hand, QuickChick's deriver is much faster; it is also better established, with more stable and predictable behavior. On the other hand, Palamedes is more flexible, as it does not require predicates to be expressed in the rigid format of inductive relations. Predicates like `List.length xs = k`, which Palamedes can synthesize a generator for directly, would need to be re-encoded before QuickChick's deriver could apply. Focusing on executable predicates also means that Palamedes applies more naturally outside of theorem provers (see Section 9). Additionally, Palamedes works harder to avoid backtracking. In cases like the STLC example discussed in Section 6.2, QuickChick derives backtracking generators without warning the user, whereas Palamedes avoids backtracking more aggressively and warns the user if it fails to eliminate it. Ultimately, we hope that Lean's ecosystem eventually includes both kinds of automation—QuickChick-style derivers for inductive relations (maybe with better backtracking control) and Palamedes for executable predicates.

Isabelle's implementation of QuickCheck provides an automated method for deriving enumerators for preconditions [8, 9]. The approach is a spiritual precursor to QuickChick's, focusing on a syntactic subset of Isabelle that can be represented as Horn clauses. One important design difference between Isabelle's approach and Palamedes is that the generator is never materialized in Isabelle; the deriving mechanism does not allow the user to inspect, modify, or tune the enumerators.

The most recent addition to this space is Cobb [26], a technique that uses program repair to turn incomplete generators into complete ones. Cobb can handle some predicates that Palamedes does not, specifically lists with unique or duplicate entries and red-black trees. However, for predicates that both approaches can handle, Palamedesis much faster. Cobb takes over 200 seconds to synthesize a BST generator, compared to Palamedes's less than 2 seconds. Cobb also assumes that users already have at least a sketch (i.e., the basic control flow) of the generator that they want, which requires more up-front effort.

Deductive Program Synthesis. Deductive program synthesis takes a specification in some logic and a set of inference rules, and searches for a proof that the specification holds; steps in the proof immediately translate to a program proved correct by the proof. The simplest version of deductive synthesis is *type-directed synthesis* [15, 19, 36, 37], where the target proof tree is an application of typing rules. This has been extended to expressive type systems like refinement types [43] and semantic types [18]. Several systems extend this approach to other logics [12, 25, 46], including separation logic [13, 23, 44].

Palamedes builds on the extensive body of work in deductive synthesis, particularly on SuSLik [44] and its followup work [13, 23, 55]. Our use of anamorphisms to build generators is closely related Hong and Aiken [20]'s use of paramphisms for synthesizing recursive algorithms. However, Palamedes targets a domain that has not yet been explored by prior work, and its implementation is different in key ways. Unlike prior work, Palamedes relies on Lean for many aspects of proof search; for example, avoiding explicit reasoning about termination and solving auxiliary lemmas with tools like Aesop [30]. Additionally, with the exception of Fiat [12] and SuSLik [55], prior work does not produce mechanized proofs of correctness. Palamedes's deduction rules are proved correct within the Lean theorem prover, and those proofs are combined to produce a proof that the final generator is correct. This strengthens the guarantee that a resulting program is correct by construction.

9 CONCLUSION AND FUTURE WORK

Our work addresses the constrained generator synthesis problem, offering an algorithm for synthesizing generators that are sound and complete with respect to a predicate, including generators for recursive data structures like lists and trees. Our approach combines prior work on deductive synthesis, functional programming, theorem provers, and more into a technique has the potential to significantly advance PBT automation.

We close with some ideas for future work.

Tuning Generator Distributions. The generators produced by Palamedes are guaranteed to produce the right set of values, but they may sample from those values with a suboptimal *distribution*. For example, the predicate $a = 1 \vee a = 2 \vee a = 3 \vee a = 4$ will yield the generator

```
pick (pure 1) (pick (pure 2) (pick (pure 3) (pure 4)))
```

which will produce `1` with probability 0.5, `2` with probability 0.25, and `3` and `4` with probability 0.125. This bias towards `1` is probably not something the user wants: it is simply an artifact of operator associativity.

Luckily, there are ways to address this problem. As a naïve solution, we could implement an optimization pass that re-associates `picks` to prefer more balanced trees. For some practical cases, this might actually be sufficient. But we need not stop there: recent work has shown that probabilistic programming languages, in particular Loaded Dice, can be used to automatically tune generators to user-specified distributions [54]. We plan to implement a translation from our generators into Loaded Dice, giving users comprehensive control over generator distributions. (Along the way, we hope to improve on the way Loaded Dice deals with recursion—currently it unrolls loops before training, but we suspect that the extra structure provided by unfolds will enable a more robust approach.)

Correct Generators for Everyday Developers. Implementing Palamedes as a Lean library has myriad benefits, but it has one major downside: theorem provers are relatively inaccessible to everyday software developers, so the tool in its current form is unlikely to see broad adoption. However, we see a clear path towards impact in the software engineering industry, by using Palamedes as the backend of more user-focused tools. As a first step, we intend to target Python. We plan to (1) embed a subset of Python's semantics in Lean, (2) compile Python predicates to that sub-language, (3) synthesize appropriate generators, and (4) render the synthesized generators as Hypothesis [31] strategies. If successful, we hope to push this paradigm even further, using Palamedes as a backend for synthesizing generators for other languages and PBT frameworks.

Better Automation and Algorithmic Improvements. As we demonstrate in Section 6, Palamedes is already flexible enough to synthesize a wide range of generators, but the algorithm may be further improved by ongoing improvements to Lean's proof automation. For example, Lean now has tactics for automating proof search with SMT [35] and e-graph rewriting [48]. Large-language-model-based proof automation is an active area of study [14, 24, 29, 53, 59, 60].

These approaches dovetail nicely with our current infrastructure—they could be used to implement more powerful versions of the `match_*` tactics, and some could even to replace Aesop entirely as the engine for the core synthesis loop. We expect Palamedes to grow naturally in power over time, working in concert with the Lean community's improvements in proof automation.

REFERENCES

[1] 2023. How We Built Cedar with Automated Reasoning and Differential Testing. https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing.

[2] Thorsten Altenkirch and Conor McBride. 2006. Towards Observational Type Theory. (2006). https://personal.cis.strath.ac.uk/conor.mcbride/ott.pdf

[3] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (ERLANG '06)*. Association for Computing Machinery, New York, NY, USA, 2–10. https://doi.org/10.1145/1159789.1159792

[4] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR Software with QuickCheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

[5] Jeremy Avigad, Mario Carneiro, and Simon Hudon. 2019. Data Types as Quotients of Polynomial Functors. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:19. https://doi.org/10.4230/LIPIcs.ITP.2019.6 ISSN: 1868-8969.

[6] R. S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica* 21, 3 (Oct. 1984), 239–250. https://doi.org/10.1007/BF00264249

[7] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 836–850. https://doi.org/10.1145/3477132.3483540

[8] Lukas Bulwahn. 2012. 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

[9] Lukas Bulwahn. 2012. 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

[10] Koen Claessen, Jonas Duregård, and Michal H. Pałka. 2015. Generating Constrained Random Data with Uniform Distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

[11] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[12] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 689–700. https://doi.org/10.1145/2676726.2677006

[13] Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 164 (June 2023), 24 pages. https://doi.org/10.1145/3591278

[14] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. https://doi.org/10.48550/arXiv.2303.04910 arXiv:2303.04910 [cs].

[15] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

[16] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24, Vol. 187)*. Association for Computing Machinery, Lisbon, Portugal, 1–13. https://doi.org/10.1145/3597503.3639581

[17] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 128:89–128:113. https://doi.org/10.1145/3563291

[18] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-directed program synthesis for RESTful APIs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 122–136. https://doi.org/10.1145/3519939.3523450

[19] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle,

Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2491956.2462192

[20] Qiantan Hong and Alex Aiken. 2024. Recursive Program Synthesis using Paramorphisms. *Proc. ACM Program. Lang.* 8, PLDI (June 2024), 151:102–151:125. https://doi.org/10.1145/3656381

[21] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing noninterference, quickly. *SIGPLAN Not.* 48, 9 (Sept. 2013), 455–468. https://doi.org/10.1145/2544174.2500574

[22] Graham Hutton. 1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* 9, 4 (July 1999), 355–372. https://doi.org/10.1017/S0956796899003500

[23] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

[24] Saketh Ram Kasibatla, Arpan Agarwal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2024. Cobblestone: Iterative Automation for Formal Verification. https://doi.org/10.48550/arXiv.2410.19940 arXiv:2410.19940 [cs].

[25] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

[26] Patrick LaFontaine, Zhe Zhou, Ashish Mishra, Suresh Jagannathan, and Benjamin Delaware. 2025. We've Got You Covered: Type-Guided Repair of Incomplete Input Generators. https://doi.org/10.48550/arXiv.2504.06421 arXiv:2504.06421 [cs].

[27] 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. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), 114–129.

[28] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[29] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. 2024. A Survey on Deep Learning for Theorem Proving. https://doi.org/10.48550/arXiv.2404.09939 arXiv:2404.09939 [cs].

[30] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 253–266. https://doi.org/10.1145/3573105.3575671

[31] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[32] Grant Reynold Malcolm. 1990. Algebraic data types and program transformation. (1990).

[33] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of functional programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

[34] Agustín Mista and Alejandro Russo. 2021. Deriving Compositional Random Generators. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3412932.3412943

[35] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. 2025. Lean-SMT: An SMT tactic for discharging proof goals in Lean. https://doi.org/10.48550/arXiv.2505.15796 arXiv:2505.15796 [cs].

[36] Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2019)*. Association for Computing Machinery, New York, NY, USA, 64–76. https://doi.org/10.1145/3331554.3342608

[37] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. *ACM SIGPLAN Notices* 50, 6 (June 2015), 619–630. https://doi.org/10.1145/2813885.2738007

[38] Micha\l H. Pa\lka, 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. https://doi.org/10.1145/1982595.1982615

[39] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing Correctly with Inductive Relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 966–980. https://doi.org/10.1145/3519939.3523707

[40] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Christian Urban and

Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

[41] Alberto Pardo. 2003. Generic Accumulations. In *Generic Programming: IFIP TC2 / WG2.1 Working Conference Programming July 11–12, 2002, Dagstuhl, Germany*, Jeremy Gibbons and Johan Jeuring (Eds.). Springer US, Boston, MA, 49–78. https://doi.org/10.1007/978-0-387-35672-3_3

[42] Alberto Pettorossi, Enrico Pietropoli, and Maurizio Proietti. 1993. The Use of the Tupling Strategy in the Development of Parallel Programs. In *Parallel Algorithm Derivation and Program Transformation*, Robert Paige, John Reif, and Raplh Watcher (Eds.). Springer US, Boston, MA, 111–151. https://doi.org/10.1007/978-0-585-27330-3_4

[43] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *SIGPLAN Not.* 51, 6 (June 2016), 522–538. https://doi.org/10.1145/2980983.2908093

[44] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the Synthesis of Heap-Manipulating Programs. *SuSLik, Tool Implementation for Article: Structuring the Synthesis of Heap-Manipulating Programs* 3, POPL (Jan. 2019), 72:1–72:30. https://doi.org/10.1145/3290385

[45] Jacob Prinz and Leonidas Lampropoulos. 2023. Merging Inductive Relations. *Reproduction Package for "Merging Inductive Relations"* 7, PLDI (June 2023), 178:1759–178:1778. https://doi.org/10.1145/3591292

[46] Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 65 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133889

[47] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1410–1421. https://doi.org/10.1145/3377811.3380399

[48] Marcus Rossel and Andrés Goens. 2024. Bridging Syntax and Semantics of Lean Expressions in E-Graphs. https://doi.org/10.48550/arXiv.2405.10188 arXiv:2405.10188 [cs].

[49] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. 2015. Type Targeted Testing. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 812–836. https://doi.org/10.1007/978-3-662-46669-8_33

[50] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7 (2023). https://doi.org/10.1145/3607860

[51] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3540250.3549139

[52] Rocq Team. 2025. Welcome to a World of Rocq. https://rocq-prover.org.

[53] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. https://doi.org/10.48550/arXiv.2412.14063 arXiv:2412.14063 [cs].

[54] Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd Millstein, Benjamin C. Pierce, and Guy Van Den Broeck. 2025. Tuning Random Generators. https://rtjoa.com/papers/draft-tuning.pdf

[55] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.* 5, ICFP, Article 84 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473589

[56] John Wrenn, Tim Nelson, and Shriram and Krishnamurthi. 2021. Using Relational Problems to Teach Property-Based Testing. *The art science and engineering of programming* 5, 2 (Jan. 2021). https://doi.org/10.22152/programming-journal.org/2021/5/9

[57] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3639121 arXiv:2308.04748 [cs].

[58] Li-yao Xia. 2021. generic-random. //hackage.haskell.org/package/generic-random

[59] Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. 2024. DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. https://doi.org/10.48550/arXiv.2408.08152 arXiv:2408.08152 [cs].

[60] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. https://doi.org/10.48550/arXiv.2306.15626 arXiv:2306.15626 [cs].

[61] Zhixuan Yang and Nicolas Wu. 2022. Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes. https://doi.org/10.48550/arXiv.2202.13633 arXiv:2202.13633 [cs].

A GREATER THAN AND LESS THAN

$$\text{def greaterThan } (n : \mathbb{N}) := \text{arbNat} \gg\!= \lambda\ lo \Rightarrow \text{pure } (lo + 1 + n)$$

Lemma A.1 (Greater Than Support). *$a \in [\![\text{greaterThan } lo]\!] \iff a > lo$.*

$$\frac{}{\Gamma \vdash \text{greaterThan } lo : \text{Gen}_{\mathbb{N}}\ (\lambda a \Rightarrow a > lo)}\ \textsc{S-GreaterThan}$$

$$\text{def lessThan } (\text{hi} : \mathbb{N}) := \text{choose } 0\ (\text{hi} - 1)$$

Lemma A.2 (Less Than Support). *If $0 < hi$, then $a \in [\![\text{lessThan } hi]\!] \iff a < hi$.*

$$\frac{\Gamma \vdash 0 < hi}{\Gamma \vdash \text{lessThan } hi : \text{Gen}_{\mathbb{N}}\ (\lambda a \Rightarrow a < hi)}\ \textsc{S-LessThan}$$

$$\frac{}{\Gamma \vdash \text{assume } 0 < hi \text{ inlessThan } hi : \text{Gen}_{\mathbb{N}}\ (\lambda a \Rightarrow a < hi)}\ \textsc{S-LessThanPartial}$$

B PREDICATE DEFINITIONS

```
def List.fold {α β: Type} (f : α -> β -> β) (z : β) (xs : List α) :=
  List.foldr f z xs
inductive Tree (α : Type) where
  | leaf : Tree α
  | node : (l : Tree α) -> (x : α) -> (r : Tree α) -> Tree α
def Tree.fold
    {α β : Type}
    (f : β -> α -> β -> β)
    (z : β)
    (t : Tree α) :
    β :=
  match t with
  | .leaf => z
  | .node l x r => f (Tree.fold f z l) x (Tree.fold f z r)
inductive Label where
  | low
  | high
inductive Atom where
  | atm (n : Nat) (l : Label)
inductive Stack where
  | mty
  | cons (a : Atom) (s : Stack)
  | ret_cons (pc : Atom) (s : Stack)
def Stack.fold
  {α : Type}
  (z : α)
  (f_c : Atom -> α -> α)
  (f_rc : Atom -> α -> α)
  (s : Stack) : α :=
  match s with
  | .mty => z
  | .cons x s' => f_c x (Stack.fold z f_c f_rc s')
  | .ret_cons pc s' => f_rc pc (Stack.fold z f_c f_rc s')
inductive Ty : Type where
  | unit
  | arrow (τ1 τ2 : Ty)
  deriving BEq
inductive Term : Type where
  | unit
  | var (n : Nat)
  | abs (τ : Ty) (t : Term)
  | app (t1 t2 : Term)
def Term.fold
    {α : Type}
    (z : α)
    (zn : Nat -> α)
```

```
    (f_abs : Ty -> α -> α)
    (f_app: α -> α -> α)
    (t : Term) :
    α :=
  match t with
  | .unit => z
  | .var n => zn n
  | .abs τ t' => f_abs τ (Term.fold z zn f_abs f_app t')
  | .app t1 t2 =>
    f_app (Term.fold z zn f_abs f_app t1) (Term.fold z zn f_abs f_app t2)
def isAllTwos : List Nat -> Bool
  | [] => true
  | x :: xs => x = 2 && isAllTwos xs
def isEvenLen : List α -> Bool
  | [] => true
  | _ :: xs => !(isEvenLen xs)
def isAllTwosEvenLen (xs : List Nat) : Bool :=
  isAllTwos xs && isEvenLen xs
def isIncreasingByOneAux (xs : List Nat) (prev : Nat) : Bool :=
  match xs with
  | [] => true
  | x :: xs' => x == prev + 1 && isIncreasingByOneAux xs' x
def isIncreasingByOne (xs : List Nat) : Bool :=
  isIncreasingByOneAux xs 0
def isLengthKAllTwos (k : Nat) (xs : List Nat) : Bool :=
  xs.length == k && isAllTwos xs
def isSortedBetween : List Nat -> Nat × Nat -> Bool := fun xs (lo, hi) =>
  match xs with
  | [] => true
  | x :: xs' => (lo <= x && x <= hi) && isSortedBetween xs' (x, hi)
def isTrue : List α -> Bool
  | [] => true
  | x :: xs => (fun _ => true) x && isTrue xs
def isAllTwosFold (xs : List Nat) : Bool :=
  List.fold (fun x b => x == 2 && b) true xs
def isAllTwosEvenLenFold (xs : List Nat) : Bool :=
  List.fold (fun x b => x == 2 && b) true xs = true
    ∧ List.fold (fun _ b => !b) true xs
def isEvenLenFold (xs : List α) : Bool :=
  List.fold (fun _ b => !b) true xs
def isIncreasingByOneFold (xs : List Nat) : Bool :=
  List.fold (fun x b prev => x == prev + 1 && b x) (fun _ => true) xs 0
def lengthFold (xs : List α) : Nat :=
  List.fold (fun _ b => b + 1) 0 xs
def isLengthKAllTwosFold (k : Nat) (xs : List Nat) :=
  List.fold (fun _ b => b + 1) 0 xs = k
    ∧ List.fold (fun x b => x == 2 && b) true xs
def isSortedBetweenFold (lo hi : Nat) (xs : List Nat) : Prop :=
```

```
  List.fold (fun x b s => decide (s <= x)
    && decide (x <= hi) && b x) (fun _ => true) xs lo
def isTrueFold (xs : List α) : Bool :=
  List.fold (fun _ b => b) true xs
def isAllTwos : Tree Nat -> Bool
  | .leaf => true
  | .node l x r => x = 2 && isAllTwos l && isAllTwos r
def isBST : Tree Nat -> (Nat × Nat) -> Bool := fun t ⟨lo, hi⟩ =>
  match t with
  | .leaf => true
  | .node l x r =>
    (lo <= x && x <= hi) &&
    isBST l ⟨lo, x - 1⟩ &&
    isBST r ⟨x + 1, hi⟩
def isComplete : Tree α -> Nat -> Bool := fun t n =>
  match t with
  | .leaf => n == 0
  | .node l _ r =>
    n > 0 &&
    isComplete l (n - 1) &&
    isComplete r (n - 1)
def isIncreasingByOneAux (t : Tree Nat) (prev : Nat) : Bool :=
  match t with
  | .leaf => true
  | .node l x r =>
    x == prev + 1 &&
    isIncreasingByOneAux l x &&
    isIncreasingByOneAux r x
def isIncreasingByOne (t : Tree Nat) : Bool :=
  isIncreasingByOneAux t 0
def isNonempty : Tree α -> Bool
  | .leaf => false
  | .node l _ r => true && isNonempty l && isNonempty r
def isAllTwosFold (t : Tree Nat) : Bool :=
  Tree.fold (fun bl x br => x == 2 && bl && br) true t
def isBSTFold (lo hi : Nat) (t : Tree Nat) : Bool :=
  Tree.fold
        (fun bl x br s =>
          match s with
          | (sl, sr) =>
            (decide (sl <= x) &&
              decide (x <= sr)) &&
              bl (sl, x - 1) && br (x + 1, sr))
        (fun _ => true) t (lo, hi)
def isCompleteFold (n : Nat) (t : Tree Nat) : Bool :=
  Tree.fold (fun bl _ br s => decide (s > 0) &&
  bl (s - 1) &&
  br (s - 1)) (fun s => s == 0) t n
```

```
def isIncreasingByOneFold (t : Tree Nat) : Bool :=
  Tree.fold
    (fun bl x br prev => x == prev + 1 && bl x && br x)
    (fun _ => true) t 0
def isNonemptyFold (t : Tree α) : Bool :=
  Tree.fold (fun _ _ _ => true) false t
def isBalanced : Tree Nat -> Nat -> Bool := fun t height =>
  match t with
  | .leaf => height <= 1
  | .node l _ r =>
    height > 0 &&
    isBalanced l (height - 1) &&
    isBalanced r (height - 1)
def isAVL (height lo hi : Nat) (t : Tree Nat) : Bool :=
  isBST t (lo, hi) && isBalanced t height
def isAVLFold (height lo hi : Nat) (t : Tree Nat) : Bool :=
  Tree.fold
      (fun bl x br bounds =>
        match bounds with
        | (sl, sr) => decide (sl <= x) && decide (x <= sr)
          && bl (sl, x - 1) && br (x + 1, sr))
      (fun _ => true) t (lo, hi) = true
    ∧
    Tree.fold
      (fun bl _ br h => decide (h > 0) && bl (h - 1) && br (h - 1))
      (fun h => decide (h <= 1)) t height
def isGoodNat (n : Nat) : Bool :=
  n == 0 || n == 1
def isGoodAtom : Atom -> Bool
  | .atm n _ => isGoodNat n
def isGoodStackFold (s : Stack) (n : Nat) : Bool :=
  Stack.fold (fun s => s == 0)
    (fun x acc s => isGoodAtom x && acc (s - 1))
    (fun pc acc s => isGoodAtom pc && acc (s - 1)) s n
def isGoodStack (s : Stack) (n : Nat) : Bool :=
  match s with
  | .mty => n == 0
  | .cons x s' => (n > 0 && isGoodAtom x) && isGoodStack s' (n - 1)
  | .ret_cons pc s' => (n > 0 && isGoodAtom pc) && isGoodStack s' (n - 1)
def getType (t : Term) (Γ : List Ty) : Option Ty :=
  match t with
  | .unit => pure .unit
  | .var n => Γ[n]?
  | .abs τ t => do
    let τ' ← getType t (τ :: Γ)
    pure (.arrow τ τ')
  | .app t1 t2 => do
    let τ1 ← getType t1 Γ
```

```
    let τ2 ← getType t2 Γ
    match τ1 with
    | .arrow τarg τres => do
      guard (τarg == τ2)
      pure τres
    | .unit => failure
def isWellTyped (Γ : List Ty) (t : Term) : Prop :=
  ∃ (τ : Ty), getType t Γ = τ
def isWellScoped : Term -> Nat -> Bool := fun t varCap =>
  match t with
  | .unit => true
  | .var n => n < varCap
  | .abs _ t => isWellScoped t (varCap + 1)
  | .app t1 t2 => isWellScoped t1 varCap && isWellScoped t2 varCap
def getTypeFold : Term -> List Ty -> Option Ty :=
  Term.fold
    (fun _ => pure .unit)
    (fun n Γ' => Γ'[n]?)
    (fun τ1 b Γ' => do
      let τ2 ← b (τ1 :: Γ')
      pure (.arrow τ1 τ2))
    (fun b1 b2 Γ' => do
      let τ1 ← b1 Γ'
      let τ2 ← b2 Γ'
      match τ1 with
      | .arrow τarg τres => do
        guard (τarg == τ2)
        pure τres
      | Ty.unit => failure)
def isWellTypedFold (Γ : List Ty) (t : Term) : Prop :=
  ∃ τ, getTypeFold t Γ = some τ
def isWellScopedFold (varCap : Nat) (t : Term) : Bool :=
  Term.fold
    (fun _ => true)
    (fun n s => s < n)
    (fun _ b s => b (s + 1))
    (fun b1 b2 s => b1 s && b2 s)
    t
    varCap
```

C EXTENDED TABLE OF BENCHMARKS

Predicate	Type	Time (s)	
`v = 2`	`Nat`	0.04	(0.01)
`2 = v`	`Nat`	0.04	(0.00)
`v = 2 ∨ v = 5`	`Nat`	0.08	(0.00)
`v = 2 ∨ v = 5 ∧ True`	`Nat`	0.08	(0.00)
`∃ a, a = 3 ∧ v = a + 1`	`Nat`	0.04	(0.00)
`5 <= v ∧ v <= 10`	`Nat`	0.08	(0.00)
`v > 5`	`Nat`	0.07	(0.00)
`v = 0 ∨ lo <= v ∧ v <= hi`	`Nat`	0.14	(0.00)
`isAllTwos v = true`	`List Nat`	0.84	(0.01)
`isAllTwosEvenLen v = true`	`List Nat`	2.76	(0.02)
`isEvenLen v = true`	`List Nat`	2.22	(0.02)
`isIncreasingByOne v = true`	`List Nat`	1.44	(0.01)
`List.length v = k`	`List Nat`	1.89	(0.01)
`isLengthKAllTwos k v = true`	`List Nat`	2.37	(0.01)
`isSortedBetween v (lo, hi) = true`	`List Nat`	1.72	(0.02)
`isTrue v = true`	`List Nat`	2.21	(0.01)
`isAllTwosFold v = true`	`List Nat`	0.34	(0.01)
`isAllTwosEvenLenFold v = true`	`List Nat`	2.73	(0.02)
`isEvenLenFold v = true`	`List Nat`	2.19	(0.01)
`isIncreasingByOneFold v = true`	`List Nat`	1.12	(0.01)
`lengthFold v = k`	`List Nat`	1.87	(0.01)
`isLengthKAllTwosFold k v = (true = true)`	`List Nat`	2.35	(0.08)
`isSortedBetweenFold lo hi v = (true = true)`	`List Nat`	1.34	(0.01)
`isTrueFold v = true`	`List Nat`	2.18	(0.01)
`isAllTwos v = true`	`Tree Nat`	0.61	(0.01)
`isBST v (lo, hi) = true`	`Tree Nat`	1.86	(0.01)
`isComplete v n = true`	`Tree Nat`	2.35	(0.02)
`isIncreasingByOne v = true`	`Tree Nat`	1.57	(0.02)
`isNonempty v = true`	`Tree Nat`	1.66	(0.04)
`isAllTwosFold v = true`	`Tree Nat`	0.60	(0.01)
`isBSTFold lo hi v = true`	`Tree Nat`	1.89	(0.02)
`isCompleteFold v n = true`	`Tree Nat`	2.37	(0.02)
`isIncreasingByOneFold v = true`	`Tree Nat`	1.55	(0.02)
`isNonemptyFold v = true`	`Tree Nat`	1.44	(0.02)
`isGoodStack v n = true`	`Stack`	5.39	(0.05)
`isGoodStackFold v n = true`	`Stack`	7.70	(0.11)
`isWellScoped v 0 = true`	`Term`	2.85	(0.03)
`isWellTyped Γ v`	`Term`	3.66	(0.03)
`isWellTypedFold Γ v`	`Term`	3.71	(0.03)
`isWellScopedFold v 0 = true`	`Term`	2.84	(0.03)
`lo <= v ∧ v <= hi`	`Nat`	0.07	(0.00)
`isAVL height lo hi v = true`	`Tree Nat`	34.44	(0.22)
`isAVLFold height lo hi v = true`	`Tree Nat`	34.37	(0.23)

D MANUALLY WRITTEN STLC GENERATOR

```
def genWellTyped (Γ : List Ty) : Gen Term := by
  let τ <- arbTy
  Term.unfold
    (fun (τ, Γ) => do
      pick
        (caseTy τ
          (fun () =>
            -- τ = .unit
            pure TermF.unitStep)
          (fun τ1 τ2 () =>
            -- τ = .arrow τ1 τ2
            pure (TermF.absStep $\tau$1 ($\tau$2, $\tau$1 :: Γ))))
        (if (Γ.indexesOf τ).length > 0 then
          pick
            (do
              let n <- elements (Γ.indexesOf .unit) (...)
              pure (TermF.varStep n))
            (do
              let τ' <- arbTy
              pure (TermF.appStep (.arrow τ' τ, Γ) (τ', Γ)))
        else do
          let τ' <- arbTy
          pure (TermF.appStep (.arrow τ' τ, Γ) (τ', Γ))))
    (τ, Γ)
```

E STLC GENERATORS FROM ETNA

```haskell
genTyp :: Gen Ty
genTyp = sized go
  where
    go 0 = return TBool
    go n =
      oneof
        [ go 0,
          TFun <$> go (div n 2) <*> go (div n 2)
        ]

genExpr :: Ctx -> Typ -> Gen Expr
genExpr ctx t = sized $ \n -> go n ctx t
  where
    go 0 ctx t = oneof $ genOne ctx t : genVar ctx t
    go n ctx t =
      oneof
        ( [genOne ctx t]
            ++ [genAbs ctx t1 t2 | TFun t1 t2 <- [t]]
            ++ [genApp ctx t]
            ++ genVar ctx t
        )
      where
        genAbs ctx t1 t2 = Abs t1 <$> go (n - 1) (t1 : ctx) t2

        genApp ctx t = do
          t' <- genTyp
          e1 <- go (div n 2) ctx (TFun t' t)
          e2 <- go (div n 2) ctx t'
          return (App e1 e2)

    genOne :: Ctx -> Typ -> Gen Expr
    genOne _ TBool = Bool <$> elements [True, False]
    genOne ctx (TFun t1 t2) = Abs t1 <$> genOne (t1 : ctx) t2

    genVar :: Ctx -> Typ -> [Gen Expr]
    genVar ctx t = [Var <$> elements vars | not (null vars)]
      where
        vars = filter (\i -> ctx !! i == t) [0 .. (length ctx - 1)]
```

```
Fixpoint genVar' (ctx: Ctx) (t: Typ) (p: nat) (r: list nat) : list nat :=
  match ctx with
  | nil => r
  | t'::ctx' =>
      if t = t'? then genVar' ctx' t (p + 1) (p :: r)
      else genVar' ctx' t (p + 1) r
  end.

Fixpoint genZero env tau : G (option Expr) :=
  match tau with
  | TBool =>
      bindGen arbitrary
              (fun b : bool => returnGen (Some (Bool b)))
  | TFun T1 T2 =>
      bindOpt
        (genZero (T1 :: env) T2)
        (fun e : Expr => returnGen (Some (Abs T1 e)))
  end.


Fixpoint genExpr env tau (sz: nat) : G (option Expr) :=
  match sz with
  | 0 =>
      backtrack
        [(1, oneOf_ (ret None)
          (map (fun x => returnGen (Some (Var x))) (genVar' env tau 0 [])))
        ;(1, genZero env tau)]
  | S sz' =>
      backtrack
        [(1, oneOf_ (ret None)
          (map (fun x => returnGen (Some (Var x))) (genVar' env tau 0 [])))
         ;
        (1, bindGen arbitrary (fun T1 : Typ =>
              bindOpt (genExpr env (TFun T1 tau) sz') (fun e1 : Expr =>
                bindOpt
                  (genExpr env T1 sz')
                  (fun e2 : Expr => returnGen (Some (App e1 e2))))))
         ;
         (1, match tau with
             | TBool =>
                 bindGen arbitrary (fun b : bool => returnGen (Some (Bool b)))
             | TFun T1 T2 =>
                 bindOpt
                   (genExpr (T1 :: env) T2 sz')
                       (fun e : Expr =>
                       returnGen (Some (Abs T1 e)))
                 end)]
  end.
```

F MORE SIDE-BY-SIDE GENERATOR COMPARISONS

F.1 One Or In Range

```
def genOneOrInRange (lo hi : Nat) : Gen Nat :=
  if h : decide (lo <= hi) = true then
    pick (pure 0) (choose lo hi (s_between_partial._proof_1 h))
  else
    pure 0

/-
Differences:
- Simplify proof for choose.
-/
def genOneOrInRange_manual (lo hi : Nat) : Gen Nat :=
  if h : lo <= hi then
    pick (pure 0) (choose lo hi (by omega))
  else
    pure 0
```

F.2 Complete Tree

```
def genCompleteTree (n : Nat) : Gen (Tree Nat) :=
  Tree.unfold
    (fun x =>
      if x.snd = 0 then pure TreeF.leaf
      else do
        let a <- arbNat
        pure (TreeF.node ((), x.2 - 1) a ((), x.2 - 1)))
    ((), n)

/-
Differences:
- Remove extra unit in collector.
-/
def genComplete_manual (n : Nat) : Gen (Tree Nat) :=
  Tree.unfold
    (fun height =>
      if height = 0 then
        pure TreeF.leaf
      else do
        let a <- arbNat
        pure (TreeF.node (height - 1) a (height - 1)))
    n
```

F.3 Sorted Between

```
  def genSortedBetween (lo hi : Nat) : Gen (List Nat) :=
  List.unfold
    (fun x =>
      if h : decide (x.snd.fst <= x.snd.snd) = true then
        pick (pure ListF.nil) do
          let a <- choose x.2.1 x.2.2 (s_between_partial._proof_1 h)
          pure (ListF.cons a (PUnit.unit, a, x.2.2))
      else
        pure ListF.nil)
    (PUnit.unit, lo, hi)

/-
Differences:
- Simplify proof for choose.
- Remove extra unit in collector.
-/
def genSortedBetween_manual (lo hi : Nat) : Gen (List Nat) :=
  List.unfold
    (fun (lo, hi) =>
      if h : lo <= hi then
        pick
          (pure ListF.nil)
          (do
            let a <- choose lo hi (by omega)
            pure (ListF.cons a (a, hi)))
      else
        pure ListF.nil)
    (lo, hi)
```

F.4 Length K, All Twos

```
def genLengthKAllTwos (k : Nat): Gen (List Nat) :=
  List.unfold
    (fun x =>
      if x.fst.fst = 0 then pure ListF.nil
      else pure (ListF.cons 2 ((Nat.pred x.1.1, PUnit.unit), PUnit.unit, PUnit.unit)))
    ((k, PUnit.unit), PUnit.unit, PUnit.unit)

/-
Differences:
- Remove two extra units in collector.
-/
def genLengthKAllTwos_manual (k : Nat): Gen (List Nat) :=
  List.unfold
    (fun len =>
      if len = 0 then
        pure ListF.nil
      else
        pure (ListF.cons 2 (len - 1)))
    k
```

F.5 AVL

```
def genAVL (height lo hi : Nat) : Gen (Tree Nat) :=
  Tree.unfold
    (fun x => do
      let __do_lift <-
        if x.snd.snd = 0 then pure TreeF.leaf
          else
            if Nat.pred x.snd.snd = 0 then
              if h : decide (x.snd.fst.fst <= x.snd.fst.snd) = true then
                pick (pure TreeF.leaf) do
                  let a <- choose x.2.1.1 x.2.1.2 (s_between_partial._proof_1 h)
                  pure (TreeF.node (PUnit.unit, PUnit.unit) a (PUnit.unit, PUnit.unit))
              else pure TreeF.leaf
            else
              assume (decide (x.snd.fst.fst <= x.snd.fst.snd)) fun h => do
                let a <- choose x.2.1.1 x.2.1.2 (s_between_partial._proof_1 h)
                pure (TreeF.node (PUnit.unit, PUnit.unit) a (PUnit.unit, PUnit.unit))
      match __do_lift with
        | TreeF.leaf => pure TreeF.leaf
        | TreeF.node bl x_1 br =>
          pure
            (TreeF.node (bl, (x.2.1.1, x_1 - 1), x.2.2 - 1) x_1
              (br, (x_1 + 1, x.2.1.2), x.2.2 - 1)))
    ((PUnit.unit, PUnit.unit), (lo, hi), height)
```

```
/-
Differences:
- Remove two extra units in collector.
- Nicer match on height to reduce some duplication.
- Generator is technically total now; this requires insight about the total
  number of values that can appear in a tree of height k.
-/
def genAVL_manual' (height lo hi : Nat) : Gen (Tree Nat) :=
  -- Guarantee that there are enough values in the range, given the height.
  assume (hi - lo > 2 ^ height) fun _ =>
    Tree.unfold
      (fun (lo, hi, height) => do
        match height with
        | 0 => pure TreeF.leaf
        | 1 =>
            pick (pure TreeF.leaf)
              (assume (lo <= hi) fun h => do  -- Will always succeed.
                -- Choose values so we never truncate the range to be too small.
                let a <-
                  choose
                    (lo + 2 ^ (height - 1))
                    (hi - 2 ^ (height - 1)) (by ...)
                pure (TreeF.node (lo, a - 1, height - 1) a (a + 1, hi, height - 1)))
        | height' + 1 => do
          assume (lo <= hi) fun h => do -- Will always succeed.
            -- Choose values so we never truncate the range to be too small.
            let a <-
              choose
                (lo + 2 ^ (height - 1))
                (hi - 2 ^ (height - 1)) (by ...)
            pure (TreeF.node (lo, a - 1, height - 1) a (a + 1, hi, height - 1)))
      (lo, hi, height)

/-
Differences:
- Entirely different approach.
- Relies on AVL.insert being correct.
-/
def genAVL_manual'' : Gen (Tree Nat) :=
  -- Generate list of arbitrary Nats
  let values <-
    List.unfold (fun () =>
      pick
        (pure (ListF.nil))
        (do let a <- arbNat; pure (ListF.cons a ())))
    ()
  -- Insert all values into an empty AVL tree.
  pure (List.fold (fun x t => AVL.insert t x) AVL.empty)
```