The Search for Constrained Random Generators

ANONYMOUS AUTHOR(S)

One of 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; the executable specifications used by PBT often have preconditions that input values must satisfy in order to be useful for testing, 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 searching for correct generators. To deal with recursive predicates, we rewrite the predicate as a catamorphism (i.e., a fold) and then match that with an appropriate anamorphism (i.e., unfold); 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] is an approach to software testing that bridges the gap between traditional testing and heavier-weight formal methods [53] by allowing developers to 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 would test with PBT:

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

This says that, assuming a tree t is a valid binary search tree (BST), inserting a new value x into that tree yields another valid BST. To test this property, a PBT framework uses a program called a *generator* to randomly sample hundreds or thousands of random values for x and t and check that the property holds for each pair of values. But the property is vacuously true if t is not a valid BST to start with—if the random generation procedure is not carefully designed, most generated trees will not 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 generate *all values* and *only values* that satisfy its precondition.

This problem has been studied extensively by programming languages researchers over the years [10, 17, 26, 50], with many proposed solutions that address it to some degree. But as of 2024, a study on PBT's usability in practice [16] still cited the availability of generators as a key challenge for PBT adoption. The key issue is speed: this study showed that, in practice, 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 during generation (e.g., rejection sampling, calling an SMT solver, 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 to be fast and to produce only valid values, by construction. This synthesis algorithm can run offline, allowing for much faster and more effective testing. The algorithm is based on a denotational semantics that captures the set of 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 the formula

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

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 [42, 43], our synthesis algorithm 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—a functional program—that can be used for testing.

For straight-line code, our proof rules are roughly one-to-one with common *generator combinators*—the basic functions used to build generators by hand. To dealing with recursive data structures, we use *recursion schemes*; in particular, 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 [22], 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 affords us significant benefits. First, synthesized generators come with mechanized proofs of correctness, ensuring that any generator produced by our algorithm is appropriate for testing. Next, working in Lean means the implementation of the synthesis procedure itself can be relatively simple—in fact, we use a popular proof search tactic, Aesop [29], 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 will continue to grow modularly.

In Section 2 we provide more concrete motivation and background, along with a preview of our approach. The remaining sections offer 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, meaning that the synthesized generators are correct by construction.
- We extend our synthesis algorithm to work for generators of recursive data types, leveraging intuitions from the literature on recursion schemes (Section 4). In particular, we establish a relationship between predicates written as folds and generators written as unfolds.
- We describe Palamedes, an implementation of our synthesis procedure that is embedded as a library in the Lean proof assistant (Section 5). In this context, we can achieve performant synthesis that maintains mechanized proofs of correctness, while borrowing key parts of the algorithm from Lean's existing infrastructure.
- We evaluate our approach with a suite of case studies, demonstrating that our synthesis procedure produces generators comparable to handwritten generators 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 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 addressing the problem (Section 2.2), and motivating a refinement (Section 2.3)—the constrained generator synthesis problem—which 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 we focus 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 property in Section 1, a developer might write a generator like the one in Figure 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 hand-written 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 then continuing.

While generators like this are familiar to PBT experts, a novice might struggle to come up with this generator from scratch. Moreover, a recent study of PBT users [16] found that even experts, who can in principle write generators like the one above, still see writing effective generators as a distraction from the 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. This is important for properties that have preconditions (e.g., the BST one above).
(2) **Completeness** The generator should be able to produce *any* value that satisfies the validity predicate. This ensures that the generator does not miss important parts of the input space.

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—and, in particular, finding fast solutions—is critical if we want to build PBT libraries that are usable for developers and powerful enough to find bugs in software.

2.1.1 Aside About Distributions. Readers familiar with PBT literature might have expected us to say something about the *distribution* of a 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 have a huge impact on testing performance—but because they can be handled separately. Recent work has shown that, once a generator is written, it can be tuned by external processes to achieve various desirable distributional qualities; we defer distributional concerns to such processes. See Section 9 for more.

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 some notable exceptions [25, 27], but we delay comparisons with these to Section 8, where we can go into more technical detail.)

Most of the available search procedures start with a naïve, complete PBT generator (e.g., derived from type information [33, 55]) and prune its generation space "online," during the generation process, to ensure soundness. The simplest way to prune the space is via *rejection sampling*—discarding any invalid values and retrying generation—but this results in poor testing performance for even moderately sparse predicates. More advanced mechanisms for pruning include laziness [10], Brzozowski derivatives [17], reinforcement learning [46], and constraint solving [48, 50]. Others try to search for valid inputs "from scratch," for example with the help of a large language model [54].

But there is something unsatisfying about all of these search-based approaches: `genBST`, above, does no searching at all! It produces valid inputs by construction, every time, essentially as quickly as possible. Could we automatically build generators like `genBST` that do no searching?

2.3 The Constrained Generator Synthesis Problem

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

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 will discuss what we mean by "efficient" in the next section. This version of the problem is harder, but solutions also 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); 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

While it is often standard to represent generators as sampling functions of type `Seed -> a`, 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 generators are represented by the following inductive data type:

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

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

While these constructors are simply data, they each have a standard interpretation as a procedure for sampling values (we discuss the interpretation in more detail in Section 5). The pure constructor represents a constant generator that always produces the same value. The $\gg\!=$ constructor sequences 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. It checks a boolean condition; if true, it simply calls its argument, but if false it fails, generating nothing. Finally, the indexed constructor represents an infinite family of generators, indexed by natural numbers.[1]

Next, we define the support of a generator.

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

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

These definitions follow the intuition given above and agree with the denotational semantics of generators presented in prior work [39].

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. We use `indexed` to turn this indexed family of partial generators into a total generator.

The support of this generator is precisely $\mathbb{N}$ (we prove this in our Lean development).

Example 3.2 (Partial Generator). While our ultimate goal is to avoid generators that search during generation, we need such generators to be representable in our language. Our synthesis procedure works by first producing potentially partial generators and then attempting to optimize them into total generators.

Here is an example of a generator that is partial:

```
def backtracks := 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.

---

[1] In lazy languages like Haskell this constructor is not necessary. But since we will be working in Lean (which is strict) we need this constructor to be able to represent generators of infinite sets (e.g., a generator of all natural numbers).

We can use the support of a generator to formally define what it means for a generator to be *sound*, *complete*, and *correct*.

Definition 3.2 (Soundness)**.** A generator g is *sound* with respect to a predicate φ if

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

Definition 3.3 (Completeness)**.** A generator g is *complete* with respect to a predicate φ if

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

Definition 3.4 (Correctness)**.** A generator g is *correct* with respect to a predicate φ if it is both sound and complete.

Notation 3.1 (Correct Generator)**.** The type of correct generators with respect to a predicate φ on 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 skipped in §2.3. Our gold standard for generators is demonstrated 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, which 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, the assume constructor means that the synthesis procedure can occasionally produce partial generators that backtrack more than an expert-written generator might.

To classify these suboptimal generators, we define the following notion:

Definition 3.5 (Assume Freedom)**.** A generator g is *assume-free* iff it does not use the assume constructor (including in sub-generators that it calls).

Each of our case studies in Section 6 is labeled to clarify whether or not the generator is assume-free; all are correct by construction.

3.2 Core Synthesis Algorithm

We will now outline an 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 creates the generator itself *en passant*. This approach to synthesis can be found in systems like SuSLik [43] and Synquid [42]. Concretely, we start with the statement

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

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:

$$\frac{\Gamma \vdash a' : \alpha}{\Gamma \vdash \mathsf{pure}\ a' : \mathsf{Gen}_\alpha\ (\lambda\ a \Rightarrow a = a')}\ \textsc{S-Pure}$$

$$\frac{\Gamma \vdash x : \mathsf{Gen}_\alpha\ P \qquad \Gamma \vdash y : \mathsf{Gen}_\alpha\ Q}{\Gamma \vdash \mathsf{pick}\ x\ y : \mathsf{Gen}_\alpha\ (\lambda\ a \Rightarrow P\ a \lor Q\ a)}\ \textsc{S-Pick}$$

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

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.

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 a disjunction by using pick. We can use these rules to synthesize a generator for the predicate $\lambda\, a \Rightarrow a = 1 \lor a = 2$, 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 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 our synthesizer may need to be able to discharge certain nontrivial theorems automatically. In §5 we discuss the specifics of this process, but for now we assert that any rule labeled "$\Downarrow$" can be discharged automatically.

Sometimes our synthesis rules do not apply directly to the goal as stated. For example consider the 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 need to apply the *conversion rule*:

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

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.

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

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

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

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):

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

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. By analogy with the definition of support for bind, we can define a synthesis rule for composing generators:

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

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 and 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:

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

The goal here is to generate an i that is equal to $j + 3$, where j is further 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 now to synthesize a generator for j, which we complete by the process 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 1.

The rule for $\gg\!=$ can be quite difficult to apply. Its conclusion has a very specific form, requiring an existential binding a and a conjunction that refers only to a on its left side. The Convert rule is critical here, allowing the synthesizer to logically manipulate the predicate before applying the rule; we discuss those logical manipulations 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 split that variable into cases.

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

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

Here we give rules for booleans and natural numbers; rules for other inductive types can be derived from their definitions (see Section 5). Each of these rules corresponds to a match in the final generator; they pick out a variable in the context, test it, and then choose a different generator depending on the outcome of that test. Note that these splits are *not* recursive; we deal with recursive generators in the next section.

A key benefit of this approach, and deductive synthesis in general, is that the generators we arrive at by applying these rules are correct by construction. The process that produces the generator simultaneously builds a proof that the generator's support is equivalent to the desired support. If this process finds a generator, a developer can be confident that the generator is appropriate for their testing needs.

3.3 Standard Library Functions

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

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

We can characterize its support as follows:

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

And we can give the corresponding synthesis rule:

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

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

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

This rule is valid, but it introduces the potential for the generator to fail: if $lo > hi$ when this generator is executed, it will not be able to produce a value. Critically, this generator is still correct—it is complete, and it is sound in the sense that if it produces a value then that value satisfies the given condition—but it is a bit unsatisfying. Luckily, we can usually optimize this issue away after the fact.

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

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

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

It has the following support and synthesis rules:

Lemma 3.2 (Elements Support). If $xs \neq []$, then $a \in [\![\text{elements } xs]\!] \iff a \in xs$.

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

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

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:

$$\frac{\cdots \qquad \overline{lo{:}\mathbb{N}, hi{:}\mathbb{N} \vdash \textsf{assume}\ l \leq h\ \textsf{in}\ \text{choose } lo\ hi : \text{Gen}\ (\lambda\ a \Rightarrow lo \leq a \leq hi)}}{lo{:}\mathbb{N}, hi{:}\mathbb{N} \vdash \textsf{pick}\ (\textsf{pure}\ 0)\ (\textsf{assume}\ lo \leq hi\ \textsf{in}\ \text{choose } lo\ hi) : \text{Gen}(\lambda\ a \Rightarrow a = 1 \vee lo \leq a \leq hi)}$$

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 $lo \leq hi$ first and only choose the right branch if the check succeeds. We leverage this observation by designing *optimization rules* that rewrite generators to avoid failures. The rule we need for the above case is

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

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

$$\begin{gathered}\textsf{pick}\ (\textsf{pure}\ 0)\ (\textsf{assume}\ lo \leq hi\ \textsf{in}\ \text{choose } lo\ hi) \rightsquigarrow \\ \text{if } lo \leq hi\ \text{ then } \textsf{pick}\ (\textsf{pure}\ 0)\ (\text{choose } lo\ hi) \text{ else } \textsf{pure}\ 0\end{gathered}$$

In total, we use six optimization rules to facilitate uncovering assumes and lifting them out of choices:

$$\begin{aligned}
\textsf{pure}\ v \gg\!= f &\rightsquigarrow f\ v && (1)\\
(x \gg\!= g) \gg\!= f &\rightsquigarrow x \gg\!= (\lambda a \Rightarrow g\ a \gg\!= f) && (2)\\
(\textsf{assume}\ b\ \textsf{in}\ x) \gg\!= f &\rightsquigarrow \textsf{assume}\ b\ \textsf{in}\ (x \gg\!= f) && (3)\\
x \gg\!= (\lambda\ a \Rightarrow \textsf{assume}\ b\ \textsf{in}\ (f a)) &\rightsquigarrow \textsf{assume}\ b\ \textsf{in}\ (x \gg\!= f) \qquad \text{if } a \notin \text{fv}(b) && (4)\\
\textsf{pick}\ (\textsf{assume}\ b\ \textsf{in}\ x)\ y &\rightsquigarrow \text{if } b \text{ then } \textsf{pick}\ x\ y \text{ else } y && (5)\\
\textsf{pick}\ x\ (\textsf{assume}\ b\ \textsf{in}\ y) &\rightsquigarrow \text{if } b \text{ then } \textsf{pick}\ x\ y \text{ else } x && (6)
\end{aligned}$$

Rules (1) and (2) are standard monad equivalences, rules (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, e.g., when there are incompatible assumptions on the two sides of a pick. But, for most of the examples in Section 6, they are enough to create a generator that is assumeFree.

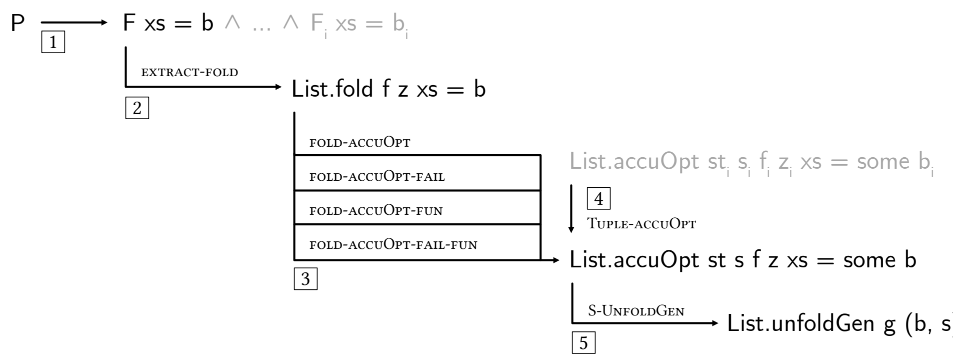

Fig. 3. How a generator is synthesized from a user-written recursive predicate over lists.

4 SYNTHESIZING GENERATORS FOR RECURSIVE STRUCTURES

We next describe how our synthesis procedure handles predicates over inductive data structures like lists and trees. Since our approach is based on *recursion schemes*, we start with some background on those (Section 4.1). Then we outline our synthesis procedure at a high level (Section 4.2), building up to a fully detailed picture of generators for recursive data (Section 4.3). Finally, we discuss the end-to-end process in a bit more detail (Section 4.5).

At a high level, the approach in this section takes some recursive predicate, transforms it into a normal form, and then applies a synthesis rule like the ones from the previous section. The pipeline that we implement is shown in Figure 3. While the details in this section are important, there is one key takeaway: 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—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.

4.1 Background: Recursion Schemes

Recursive functions are a common challenge for program analysis and synthesis tasks, even in strongly normalizing languages where they are guaranteed to terminate. While there are techniques available for synthesizing recursive programs directly from recursive specifications [42, 43], we take a different approach that is theoretically simpler and easier to embed in Lean.

In the functional programming community, there is 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.

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

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

This function takes as arguments a "base-case" z and a "step function" f. We call the type β the "collector" for the fold.[2] When the list is empty, we return z. When the list is a cons-cell, we

[2]Other texts call this value the "accumulator," but we use "accumulation" to refer to a type of fold [40] and we want to avoid confusion.

recursively call List.fold f z on the tail of the list and then 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.[3] We first compute something about the tail of the list, without considering the value at the head, and only at the end do we actually consider the information at the head. This will be useful to remember later.

We can use List.fold to implement common functions over lists, for example length:

$$\text{def length } xs := \text{List.fold } (\lambda\ x\ b \Rightarrow 1 + b)\ 0\ xs$$

4.1.2 More Advanced Recursion Schemes. While List.fold can represent all terminating functions over lists [21], 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 [58]. Of note for this paper is the *accumulation* [40], which we present with some minor simplifications:

```
def List.accu (st : α → σ → σ) (s : σ) (f : α → β → σ → β) (z : σ → β) (xs : List α) : β :=
  match xs with
  | [] ⇒ z s
  | x :: xs′ ⇒ f x (List.accu st (st x s) f z xs′) s
```

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

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

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

The accumulation state is the minimum value allowable in the remaining segment of the list; it is initialized to 0 and updated to be the most recently seen value at each step. The step function then checks that $lo \le x$ and conjoins that with the value computed from the tail of the list (which ensures 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. This higher-order function can be difficult to work with, so accumulations provide a convenient alternative representation.

4.1.3 Failing Folds. We need versions of List.fold and List.accu that capture *failing 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
```

[3] 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.

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

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

The following function nondeterministically "unfolds" a value into a whole data structure:

```
def List.unfoldGen (g : β → Gen (Option (α × β))) (b : β) : Gen (List α) :=
  let rec go b fuel :=
    match fuel with
    | 0 ⇒ 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, the function 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'. We use the indexed constructor to unify this indexed family of generators into a single generator for lists.

A key benefit of List.unfoldGen 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.unfoldGen (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

Many real-world properties take inductive data as input, so it is important that our synthesis procedure be able to handle predicates over inductive data. The powerful tools for structuring recursion that we reviewed in the previous subsection will allow us to do just that.

For a first example, consider the following predicate, which checks that a list has a given length:

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

This uses the definition of length that we saw earlier, which relies on List.fold for recursion. We can also use List.unfoldGen to write a generator for values satisfying this predicate:

```
def genLengthK (k : ℕ) : Gen (List ℕ) :=
  List.unfoldGen
    (λ n ⇒
      match k with
      | 0 ⇒ pure none
      | 1 + k′ ⇒ arbNat ≫= λ x ⇒ pure (some (x, k′)))
    n
```

At each unfolding step, the generator checks the seed value n. If $n = 0$ then it generates none, indicating that the list should end (this makes sense, 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 hasLengthK? The key observation is that hasLengthK and genLengthK have an inverse relationship—whenever genLengthK takes a step, it is guaranteed that hasLengthK can undo that step. We can make that observation concrete with the following propositions:

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

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

We can state a more general version of this relationship as a lemma.

Lemma 4.1 (Fold-UnfoldGen-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 relationship holds of the unfold and fold

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

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." 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 while ensuring that the fold would always compute the same information going the other way.

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

$$\frac{\Gamma \vdash g : (b : \beta) \to \mathsf{Gen}_{\mathsf{Option}\ (\alpha\times\beta)}\ (P\ b)}{\Gamma \vdash \mathsf{List.unfoldGen}\ g\ b : \mathsf{Gen}_{\mathsf{List}\ \alpha}\ (\lambda\, xs \Rightarrow \mathsf{List.fold}\ f\ z\ xs = b)}\ \textsc{S-UnfoldGen'}$$

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

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

4.3 Handling More Complex Folds

The S-UnfoldGen' rule works for predicates whose folds have an exact match as an unfolds, but other folds require a pre-processing 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-UnfoldGen', we could turn this directly into a generator of the form:

$$
\begin{array}{l}
\mathsf{List.unfoldGen} \\
\quad (\lambda\, b \Rightarrow \\
\quad\quad \mathsf{match}\ b\ \mathsf{with} \\
\quad\quad \mid \mathsf{true} \Rightarrow \mathsf{pick}\ (\mathsf{pure}\ \mathsf{none})\ (\mathsf{pure}\ (\mathsf{some}\ (2, \mathsf{true}))) \\
\quad\quad \mid \mathsf{false} \Rightarrow \cdots \\
\quad \mathsf{true},
\end{array}
$$

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 false branch at all.

The key observation is that allTwo has a hidden invariant that the S-UnfoldGen' 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 a failing fold:

$$
\begin{array}{l}
\mathsf{def\ allTwo}\ (xs : \mathsf{List\ Nat}) := \\
\quad \mathsf{List.foldOpt}\ (\lambda\, x\ ()\ \Rightarrow \mathsf{if}\ x = 2\ \mathsf{then\ some}\ ()\ \mathsf{else\ none})\ (\mathsf{some}\ ())\ xs = \mathsf{some}\ ()
\end{array}
$$

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:

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

$$
\begin{array}{ll}
\text{where} & P\ b = \\
& \quad \lambda\, step \Rightarrow \\
& \quad\quad (step = \mathsf{none} \land z = \mathsf{some}\ b) \lor \\
& \quad\quad (\exists\, x\ b',\ step = \mathsf{some}\ (x, b') \land f\ x\ b' = \mathsf{some}\ b)
\end{array}
$$

This rule looks roughly the same as S-UnfoldGen', 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:

$$
\begin{array}{l}
\mathsf{def\ genAllTwo} : \mathsf{Gen\ (List\ Nat)} := \\
\quad \mathsf{List.unfoldGen}\ (\lambda\, ()\ \Rightarrow \mathsf{pick}\ (\mathsf{pure\ none})\ (\mathsf{pure}\ (\mathsf{some}\ (2, ()))))\ ()
\end{array}
$$

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

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

$$
\begin{array}{ll}
\text{where} & P\ b\ s = \\
& \quad \lambda\, step \Rightarrow \\
& \quad\quad (step = \mathsf{none} \land z\ s = \mathsf{some}\ b) \lor \\
& \quad\quad (\exists\, x\ b',\ step = \mathsf{some}\ (x, b') \land f\ x\ b'\ s = \mathsf{some}\ b) \\
& g'\ b\ s = \\
& \quad g\ b\ s \gg\!= \lambda\, mstep \Rightarrow \\
& \quad \mathsf{match}\ mstep\ \mathsf{with} \\
& \quad \mid \mathsf{none} \Rightarrow \mathsf{pure\ none} \\
& \quad \mid \mathsf{some}\ (x, b') \Rightarrow \mathsf{pure}\ (\mathsf{some}\ (x, (b', st\ x\ s)))
\end{array}
$$

This it allows for both a collector that passes information backward from the tail of the list and a state that passes data forward, and it allows for partiality.

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 ca use S-UnfoldGen to derive:

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

This generator is behaves the same as one an expert user might write. At each step, the generator 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-UnfoldGen rule works for predicates that are written as a single pass over a data structure, but sometimes predicates have multiple independent constraints. For example,

$$\text{def allTwoLengthN } (k : \mathbb{N})\ (xs : \text{List Nat}) := \text{allTwo } xs \land \text{hasLengthK } k\ xs$$

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

We have two options for handling these situations. The first is to draw from the literature on program calculation and apply a *tupling* transformation [6, 41] to combine the conjuncts. These transformations were designed for functional program optimization, but they are a natural fit for this problem. Another is to try to adapt the "merging" procedure for inductive relations described by Prinz and Lampropoulos [44]. It turns out that, at 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 that computes $P\ xs \land Q\ xs$. The definiton 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 would need to apply manually in systems like QuickChick [44].

It is also worth pointing out that other tupling is not the only useful program transformation that the early program calculation literature proposed for recursive programs. If we find that other transformations (e.g., fusion) are useful for the kinds of predicates that users care about, we could add those steps to the pipeline.

4.5 Putting it All Together

The previous sections have introduced a lot of machinery; now we put that machinery together into a standardized workflow for handling predicates on inductive data.

The workflow is presented at a high level in Figure 3.

(1) Rewrite 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) If possible, rewrite each F as a fold. Concretely, search for z and f satisfying the following equations.

$$F\ [] = z$$
$$F\ (x :: xs) = f\ x\ (F\ xs)$$

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" [21, 31].

(3) Rewrite each fold as a failing 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 different branches of the predicate together.

(5) Apply S-UnfoldGen 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.

4.6 Other Inductive Types

Everything in this section so far has revolved around the List data type, but there is actually nothing in the above workflow that is specific to lists. All recursive data types that are composed of products and sums admit operations that are analogous to List.fold, List.accuOpt, etc.[4] This means that the pipeline from Figure 3 can be directly generalized to the kinds of data structures that everyday programmers use.

The case studies in Section 6 required us to implement this pipeline (along with other utilities, e.g. for case splitting) for 5 data structures: lists, binary trees, STLC types, STLC terms, and stacks (from [20]). The process is mechanical, as the definitions follow the structure of the data type and its constructors. We believe it will be straightforward to automate via meta-programming or by leveraging *quotients of polynomial functors* (QPFs) [5]. We leave this engineering exercise as future work, as it does not change the feasibility of our approach.

5 PALAMEDES: SYNTHESIZING GENERATORS IN LEAN

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

5.1 Synthesizer as Library

Palamedes is implemented as a library in the Lean theorem prover. Because of Lean's powerful meta-programming capabilities, this does not require compiler plugins or additional build steps as other synthesizers might: the synthesizer is implemented in standard Lean code.

Starting with a definition of the BST 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,
```

[4] Technically speaking, these functions exist for all algebraic data types that arise as the least fixed-point of a traversable functor [32].

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)
```

The user defines some predicate that they want their data to satisfy (such as isBST above), then they use the `generator_search` tactic to find an efficient generator that is sound and complete with respect to that predicate. The above call to `generator_search` finds the following generator:[5]

```
def genBST (lo hi : Nat) : Gen (Tree Nat) :=
  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. As we discuss later in this section, the above code can be pasted directly into the user's file; from there, they can modify it to tweak the distribution and make any other changes that they see fit.

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. This is one of the major benefits of working inside a theorem prover like Lean—we do not need to rely on meta-arguments that our synthesis procedure is correct. Each synthesized generator comes with mechanized proofs that the generator is indeed appropriate for the user's testing needs. (We provide 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` ($\text{Gen}_\alpha\ P$ from above). It is implemented in Lean as:

```
def CorrectGen {α : Type} (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 that value satisfies a given predicate. A value of type `CorrectGen P` is therefore a pair of a generator and a proof that that generator's support is equivalent to `P`. Next, we optimize the generator with the rewrites described in Section 3.4; this procedure also produces a `CorrectGen`, ensuring that optimization has not changed the support of the generator. Finally, we prove that the generator is assume-free (Definition 3.5). We now describe these steps in more detail.

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

5.2.1 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 [29]. Aesop takes a large list of Lean tactics and applies them in a loop; when all tactics fail to solve a particular sub-goal, the search backtracks to try a different route. Relying on Aesop in this way turned out to be extremely effective both in terms of results (in our case studies) and in terms of ease of implementation.

The rules we provide to Aesop mirror the ones described in Section 6 are given in Table 1. The core synthesis rules each apply a function that builds a term of type `CorrectGen`; 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 that the rule is valid with respect to the definition of the generator's support (elided in the above definition). 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 instead combine it with the various generator builders. For example,

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

uses `convert` with a pre-determined end goal (the conclusion of `s_pick`). This tactic tries to apply `s_pick`, using `match_pick` to prove that the rule actually applies.

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 applying the `convert` version allows `match_pick` to 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, 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* similarly to `s_bind`. The tactic

```
split_cases 1 Nat.split
```

for example, looks for the second `Nat` in the context and attempts to split it with S-SplitNat.

5.2.2 Step 2: Optimize. The optimization 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), 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 (it is straightforward to prove on paper). Instead, we use proof automation to show that each optimized generator is equivalent to its unoptimized counterpart on a case-by-case basis.

5.2.3 Step 3: (Optionally) Prove Assume-Free. While Palamedes does not categorically reject generators that contain assumes (they can still be useful in some cases), we would prefer to prove that generators are assume free. We use straightforward proof automation to attempt to prove that a generator does not make nontrivial use of the assume constructor; if we fail to prove this fact, we output a warning for the user.

5.2.4 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. Rendered generators are static, which means they are not re-synthesized when the file loads, and they can be further manipulated and tuned by the user. The user might, for example, want to add weights that bias the distribution or add size bounds to ensure generated values do not get too big.[6] When the user wants to render a generator, they can use a variant of the search tactic: `generator_search?`. This version does the same synthesis procedure and then provides the user with a "try this" widget [7] in their editor. Clicking on the hint pastes the full text of the generator into their file, which 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.

5.2.5 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 the case of failure (e.g., from assume), and functions wrapped by indexed are given increasing fuel after running out.

6 EVALUATION

In this section, we evaluate our synthesis algorithm by using Palamedes to synthesize over 40 benchmark generators. Our synthesis procedure is not complete (see Section 7), but we demonstrate that it works for a wide range of interesting and useful predicates.

We answer 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 answer 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 wide range of predicates. Some demonstrate specific aspects of our synthesis algorithm (including low-level examples that appear in the text above), while others are drawn from the literature on PBT [25, 27, 44].

A selection of the benchmark predicates are presented in Table 2; the rest 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 (AVL trees) containing multiple predicates that need to be tupled together. It takes under 2 seconds so synthesize `genBST` and under 8 seconds to get a generator for well-typed STLC terms.

We consider these synthesis times to be well within the acceptable range for a procedure like this. The most direct comparison in this regard is Cobb [25], which also uses program synthesis to obtain generators. While Cobb can synthesize some generators that Palamedes cannot yet (see Section 7) and vice versa, for the generators that they both synthesize Palamedes is much faster. Cobb takes

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

[7] 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 16GB 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	(0.00)
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	0.15	(0.00)
allTwos **v** = true	List Nat	0.84	(0.01)
allTwosEvenLen **v** = true	List Nat	3.60	(0.02)
evenLen **v** = true	List Nat	2.36	(0.02)
increasingByOne **v** = true	List Nat	1.44	(0.01)
List.length **v** = k	List Nat	1.89	(0.01)
lengthKAllTwos k **v** = true	List Nat	2.29	(0.02)
sortedBetween **v** (lo, hi) = true	List Nat	1.72	(0.01)
constTrue **v** = true	List Nat	2.34	(0.02)
allTwosTree **v** = true	Tree Nat	0.61	(0.01)
isBST **v** (lo, hi) = true	Tree Nat	1.87	(0.01)
isCompleteTree **v** n = true	Tree Nat	2.37	(0.03)
increasingByOneTree **v** = true	Tree Nat	1.58	(0.01)
nonempty **v** = true	Tree Nat	3.08	(0.04)
isGoodStack **v** n = true	Stack	5.24	(0.08)
isWellScoped **v** 0 = true	Term	2.69	(0.04)
wellTyped Γ **v**	Term	7.20	(0.10)
lo <= **v** ∧ **v** <= hi	Nat	0.07	(0.00)
isAVL height lo hi **v** = true	Tree Nat	34.51	(0.32)

over 200 seconds to synthesize a BST generator, compared to Palamedes's less than 2 seconds. Additionally, our assumption, based on prior work [16], is that developers often run their tests far more frequently than they change their definitions. That means that these modest synthesis costs will be amortized across many test runs. Critically, any costs incurred by search-based approaches to generation are paid every time the developer runs their tests.

The vast majority of synthesized generators are proved assume-free; only three—two versions of AVL trees and a demonstration example—contain assume after optimization. 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 for a node before that branch is deep enough to be balanced. Indeed, AVL tree generator that Prinz and Lampropoulos [44] present in their paper on merging inductive relations also backtracks in this relatively unlikely case. In principle, it may be possible for Palamedes to rewrite the AVL specification and produce an assume-free generator, but it is unclear how to do that automatically.

6.2 Synthesized vs. Handwritten Generators

Now we focus on **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 synthesized, which is shown in Appendix D. Following the S-Bind rule, `genWellTyped` pick a random type and then generates a random term of that type—this happens to be the method that prior work [37] 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` that determines the next data constructor that should be generated. In both cases (unit or arrow), the generator might choose to select a variable from the context, generate a function application, or generate a type-specific term—unit 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 of generation.

This generator is logically the same as one that an expert user might write to satisfy the same predicate, although the code is a bit different. In Appendix E, we present a version of this generator that the authors wrote manually for comparison. The manual generator is shorter, mostly because of reduced repetition. Rather than split on τ right away, the manual generator does it later, eliminating clauses that should appear no matter the type of the term being generated. Also, the synthesized generator has some artifacts from the synthesis process, like the match statement at the end, that an expert user would naturally eliminate. Still, the asymptotic performance of the generators is the same—neither backtracks, and both contain the same basic clauses. Both generators are ideal for testing.

We can also compare this generator to ones from the literature, in particular the STLC generators from the Etna PBT evaluation framework [49]. We replicate both the Haskell and Rocq versions of the generator in Appendix F. Focusing on the Haskell version first, there are a few key differences: Etna's version uses Haskell's size mechanism, which Palamedes does not yet have, and some other Haskell features that simplify the code. But at its core, tracing the control flow of Etna's `genExpr` yields essentially the same structure as Palamedes's `genWellTyped`. Interestingly, the QuickChick version of the STLC generator in the Etna suite, which is derived from inductive relations [27], 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 backtrack. This is not terribly wasteful, but it is sub-optimal.

7 LIMITATIONS

We think our approach is a critical step on the path to solving the constrained generator synthesis problem, but it does of course have some limitations.

First, our synthesis algorithm is necessarily partial; there are many Lean predicates that Palamedes cannot handle. For example, at the time of writing, Palamedes's tupling and fold normalization pipeline is not powerful enough to synthesize a generator for red-black trees [27]. Additionally, our choice to avoid assume at all costs means that there are some generators (e.g., "Unique List" [25]) that would be very difficult for us to synthesize with the current set of base generators. Luckily, Palamedes is extremely extensible, so these limitations can be mitigated over time. New tools for proof automation may make it easier to synthesize the RBT generator, and new library generators could unlock Unique List. In Section 9, we discuss plans to extend the set of predicates that Palamedes can handle in a variety of ways.

There is also a classic PBT generator feature that Palamedes has not yet addressed: *sizes*. QuickCheck [11] generators 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 [39] gives a semantics for the sized and resize constructors that plays well with our definitions—but we do not yet have a heuristic for how the synthesizer should use those constructors. That said, there are two ways to control the sizes of generated values in Palamedes. Users can augment their specifications with explicit size constraints. For example, `lengthKAllTwos` 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 modify their generators after synthesis to add size control.

8 RELATED WORK

In this section we cover related work in constrained random generation, generator synthesis, and deductive synthesis more generally.

8.1 The Constrained Random Generation Problem

As we discuss at length in Section 2.3, there have been many approaches to the constrained random generation problem proposed over the years [10, 17, 46, 48, 50, 54]. 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 an appropriate generator ahead of time. In situations where a developer writes their generator once and then runs it many times—which is common in PBT [16]—these approaches are likely not to scale as well as approaches that synthesize generators ahead of time.

8.2 The Constrained Generator Synthesis Problem

As discussed in the introduction, generating inputs that satisfy a given constraint is a well explored area of research. 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 [51], QuickChick provides an automated mechanism for deriving generators that are sound and complete with respect to inductive relations [27, 38]. This mechanism is incredibly powerful, efficiently deriving high quality generators, but we see clear trade-offs between QuickChick's approach and ours. On one hand, QuickChick's deriver is much faster, and it is also better established with more stable and predictable behavior. On the other hand, Palamedes is more flexible; it does not require predicates to be expressed in the rigid structure of inductive relations, meaning it has the potential to apply outside of theorem provers (see Section 9). Additionally, Palamedes works harder to avoid backtracking. It tuples conjoined predicates automatically (QuickChick's merging procedure needs to be invoked manually) and in cases like the one in Appendix F it can derive backtracking generators without warning the user. Ultimately, we hope that Lean's ecosystem eventually includes both kinds of automation—QuickChick style derivers for inductive relations (maybe with tighter control around backtracking) and Palamedes for less structured predicates.

Similarly, Isabelle's 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, which can capture a portion of the inductive relation that QuickChick can handle in Rocq (as inductive relations are structurally similar to Horn clauses) as well as a portion of the functions Palamedes can handle in Lean (by turning function definitions into Horn clauses). One important design difference between Isabelle's approach and Palamedes is that the generator is never materialized; the deriving mechanism gives little ability to inspect, modify, or tune the resulting enumerators.

Finally, the most recent addition to this space is Cobb [25], a technique that uses program repair to turn incomplete generators into complete ones. Cobb works with some predicates that Palamedes currently cannot (specifically lists with unique or duplicate entries and red-black trees), and it can repair existing user-written generators rather than requiring that generators be synthesized from scratch. On the other hand, Cobb does require both generator sketches and predicate specifications, whereas Palamedes only require predicate specifications, and Palamedes is also much faster for comparable predicates. Palamedes can also handle some predicates (e.g., AVL trees and STLC terms) that Cobb's paper does not tackle.

8.3 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, 35, 36], where the target proof tree is an application of typing rules. This has been extended to more expressive type systems like refinement types [42] and semantic types [18]. Several systems extend this approach to more general logics [12, 24, 45], including separation logic [13, 22, 43].

Palamedes builds on the extensive body of work in deductive synthesis, particularly on SuSLik [43] and its followup work [13, 22]. Palamedes targets a domain that has not yet been explored by prior work, and its implementation is also different in key ways. Unlike prior work, Palamedes relies on Lean for many aspects of the proof search; for example, avoiding explicit reasoning about termination and solving auxiliary lemmas with tools like Aesop [29]. Additionally, with the exception of Fiat [12], 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

In this work, we address the constrained generator synthesis problem. We give an algorithm for synthesizing generators that are sound and complete with respect to a predicate, including generators that produce recursive data structures like lists and trees. Our approach combines prior work on deductive synthesis, functional programming, theorem provers, and more into a technique that we think will have significant positive impact in the realm of property-based testing.

In the remainder of the paper, we discuss 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 produce those values with a suboptimal *distribution*. For example, the predicate $a = 1 \lor a = 2 \lor a = 3 \lor a = 4$ 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 not something the user expressed that they wanted per se, 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 practical cases, this might actually be sufficient. But we need not stop there: recent work has shown that probabilistic programming languages, in particular one called Loaded Dice, can be used to automatically tune generators to user-specified distributions . We plan to implement a translation from our generators into Loaded Dice, giving much more 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 Everyday Developers. Implementing Palamedes as a Lean library has myriad benefits, but it has one major downside: theorem provers are still 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 will 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 [30] strategies. If successful, we hope to push this paradigm even further, using Palamedes as a backend for synthesizing generators in as many major languages and PBT frameworks as possible.

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 [34] and e-graph rewriting [47], and large-language-model-based proof automation is an active area of study [14, 23, 28, 52, 56, 57].

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 in power naturally over time, working in concert with the Lean community's commitment to 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. Palka. 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] 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

[21] 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

[22] 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

[23] 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].

[24] 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

[25] 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].

[26] 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.

[27] 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.

[28] 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].

[29] 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

[30] 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.

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

[32] 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

[33] 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

[34] 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].

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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].

[48] 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

[49] 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

[50] 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

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

[52] 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].

[53] 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

[54] 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].

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

[56] 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].

[57] 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].

[58] 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

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

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

C EXTENDED TABLE OF BENCHMARKS

Predicate	Type	Time (s)	
v = 2	Nat	0.04	(0.00)
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	0.15	(0.00)
allTwos **v** = true	List Nat	0.84	(0.01)
allTwosEvenLen **v** = true	List Nat	3.60	(0.02)
evenLen **v** = true	List Nat	2.36	(0.02)
increasingByOne **v** = true	List Nat	1.44	(0.01)
List.length **v** = k	List Nat	1.89	(0.01)
lengthKAllTwos k **v** = true	List Nat	2.29	(0.02)
sortedBetween **v** (lo, hi) = true	List Nat	1.72	(0.01)
constTrue **v** = true	List Nat	2.34	(0.02)
allTwosFold **v** = true	List Nat	0.85	(0.01)
allTwosEvenLenFold **v** = true	List Nat	3.60	(0.03)
evenLenFold **v** = true	List Nat	2.35	(0.01)
increasingByOneFold **v** = true	List Nat	1.13	(0.00)
lengthFold **v** = k	List Nat	1.90	(0.01)
lengthKAllTwosFold k **v** = (true = true)	List Nat	2.28	(0.01)
sortedBetweenFold lo hi **v** = (true = true)	List Nat	1.34	(0.01)
trueFold **v** = true	List Nat	2.35	(0.04)
allTwosTree **v** = true	Tree Nat	0.61	(0.01)
isBST **v** (lo, hi) = true	Tree Nat	1.87	(0.01)
isCompleteTree **v** n = true	Tree Nat	2.37	(0.03)
increasingByOneTree **v** = true	Tree Nat	1.58	(0.01)
nonempty **v** = true	Tree Nat	3.08	(0.04)
allTwosTreeFold **v** = true	Tree Nat	0.62	(0.00)
isBSTFold lo hi **v** = true	Tree Nat	1.94	(0.01)
isCompleteTreeFold n **v** = true	Tree Nat	2.41	(0.03)
increasingByOneTreeFold **v** = true	Tree Nat	1.59	(0.01)
nonemptyFold **v** = true	Tree Nat	1.81	(0.04)
isGoodStack **v** n = true	Stack	5.24	(0.08)
isGoodStackFold **v** n = true	Stack	7.00	(0.06)
wellTyped Γ **v**	Term	7.20	(0.10)
isWellScoped **v** 0 = true	Term	2.69	(0.04)
wellTypedFold Γ **v**	Term	11.92	(0.16)
lo <= **v** ∧ **v** <= hi	Nat	0.07	(0.00)
isAVL height lo hi **v** = true	Tree Nat	34.51	(0.32)
isAVLFold height lo hi **v** = true	Tree Nat	34.57	(0.19)

D PALAMEDES SYNTHESIZED STLC GENERATOR

```
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)
```

E 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)
```

F STLC GENERATOR FROM ETNA

```
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)]
```

G MORE SIDE-BY-SIDE GENERATOR COMPARISONS