The Search for Constrained Random Generators

HARRISON GOLDSTEIN, University at Buffalo, SUNY, USA
HILA PELEG, Technion, Israel
CASSIA TORCZON, University of Pennsylvania, USA
DANIEL SAINATI, University of Pennsylvania, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Among the biggest 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, and only values, satisfying that predicate. Efficient solutions to this problem are critical, since the executable specifications used by PBT often have preconditions that input values must satisfy in order to be valid test cases, and satisfying values are often sparsely distributed.

We propose a novel approach to this problem using deductive program synthesis. We present a set of synthesis rules, based on a denotational semantics of generators, that give rise to an automatic procedure for synthesizing correct generators. Our system handles recursive predicates by rewriting them as catamorphisms and then matching with appropriate anamorphisms; 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. The synthesis algorithm itself is built out of standard proof-search tactics, reducing implementation burden and allowing the algorithm to benefit from further advances in Lean proof automation.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: Property-based testing, program synthesis, Lean

ACM Reference Format:
Harrison Goldstein, Hila Peleg, Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C. Pierce. 2026. The Search for Constrained Random Generators. *Proc. ACM Program. Lang.* 10, PLDI, Article 251 (June 2026), 45 pages. https://doi.org/10.1145/3808329

1 Introduction

Property-based testing (PBT) [10] aims to bridge the gap between traditional testing and heavier-weight formal methods [54] by allowing developers to automatically test software systems against formal specifications. PBT has been used to find bugs in a wide range of real-world software [1–3, 6], but important challenges remain.

One key challenge is the *constrained random generation problem*. Consider a classic example of the kind of property one might test with PBT:

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

Authors' Contact Information: Harrison Goldstein, hgoldste@buffalo.edu, University at Buffalo, SUNY, Buffalo, New York, USA; Hila Peleg, hilap@cs.technion.ac.il, Technion, Haifa, Israel; Cassia Torczon, ctorczon@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Daniel Sainati, sainati@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Leonidas Lampropoulos, leonidas@umd.edu, University of Maryland, College Park, Maryland, USA; Benjamin C. Pierce, bcpierce@cis.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART251
https://doi.org/10.1145/3808329

This says that, if a tree t is a valid binary search tree (BST), then inserting a new value x into t yields another valid BST. To test this property, a PBT framework uses a program called a *generator* to randomly sample thousands of values for x and t and check that the property holds for each pair of values.

```
def isBST t lo hi :=
  match t with
  | leaf ⇒ true
  | node l x r ⇒
    lo <= x && x <= hi &&
    isBST l lo (x - 1) &&
    isBST r (x + 1) hi
```

Fig. 1. A recursive Lean predicate that checks if a tree is a BST.

Constrained random generation. Not every way of sampling x and t will be effective. In particular, the property is vacuously true if t is not a valid BST to start with, i.e., if it fails to pass the predicate in Figure 1. If the generator is not carefully designed, most trees will be discarded as invalid. The vast majority of labeled binary trees with more than one or two nodes are not ordered in a way that makes them valid BSTs, so only a few will actually be used to test the insert function. This motivates the *constrained random generation problem*: to test a conditional property effectively, we need a way to randomly sample from the set of all and only the values that satisfy its precondition.

A PBT expert might address this problem by writing a generator manually—e.g., the one in Figure 2—that produces BSTs by construction. This generator tracks the range of values that may validly appear in a given sub-tree. If 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` (this allows us to generate trees that aren't of maximum height) or generate a `node` by selecting a value in the appropriate range and recursively generating subtrees with truncated ranges. Lean's **`do`** notation sequences generators by sampling from one generator, binding the sampled value to a variable, and continuing as another generator.

```
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. 2. A handwritten generator for binary search trees.

Unfortunately, PBT novices can struggle to come up with such generators from scratch. Indeed, a recent study of PBT users [16] found that even experts, who can in principle write effective generators like `genBST`, still see writing generators as a distraction from other testing tasks. Consequently, this problem has been studied extensively by programming languages researchers over the years. Many of the approaches revolve around some kind of search procedure: start with a naïve generator (e.g., one derived from type information [34, 56]) and prune its results "online" during generation to rule out invalid values, using techniques like constraint solving [49, 50] reinforcement learning [47], laziness [9], iterated Brzozowski derivatives [17], constrained logic programming [12], and large language models [55].

But searching for valid inputs can make generators unusably slow. Many PBT users run their tests often, in a tight loop [16], and such workflows demand very efficient generation. Moreover, we know that generating valid inputs without expensive searching is possible—the `genBST` generator in Figure 2 does no searching at all! If we could *automatically* build generators that don't search, we could significantly improve testing-time performance while giving users the convenience they want.

The Constrained Generator Synthesis Problem. Following this intuition, we propose a new *constrained generator synthesis problem* that refines the familiar constrained random generation problem by requiring solutions that build generators like `genBST` "offline," rather than performing search at testing time. A couple of existing tools, in particular QuickChick [29] and Cobb [26] address this more specific version of the problem (we discuss these in detail in Section 6), but in general this version of the problem is under-explored.

We propose a novel approach to the constrained generator synthesis problem that uses a *deductive synthesis* algorithm to search for generators that are guaranteed—by construction—to produce valid values without searching at run time. The approach is based on a denotational semantics that characterizes the values that a given generator can generate. To get a generator for a property whose precondition is φ, we compute a generator g satisfying $\forall\, a,\ \ a \in [\![g]\!] \iff \varphi(a)$, 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: the generator is *sound* and *complete*. Inspired by systems like Synquid [43] and SuSLik [44], our synthesis algorithm works by building a *proof* that there exists an appropriate g for a given φ by applying a series of proof rules, then extracts a proof witness that embodies g in executable form. The resulting generators are correct by construction, since the search constructs the proof as well.

Our basic proof rules align with common *generator combinators*—the core functions used to build generators by hand. To construct recursive generators for inductively defined data structures, we use *recursion schemes*; concretely, we observe that many predicates that can be represented as a fold (or catamorphism) have an associated generator represented as an unfold (or anamorphism). The upshot is that we can deal with first- and second-order primitive-recursive predicates using pre-derived induction principles, rather than relying on cyclic proofs [23].

It is important to note that our synthesis procedure is guaranteed to produce generators with the correct *support* (i.e., the right set of values), but not necessarily optimal *distributions*. We have chosen to treat generators as nondeterministic programs, focusing on the set of values that they can produce and ignoring the probabilities used to produce them. This is a significant simplification, but an important one. Rather than try to do everything at once, we think synthesizing generators with provably optimal support and then *tuning* them is significantly more tractable and flexible than including tuning in the synthesis process. Ongoing work on generator tuning [52, 60] has made promising steps towards automating this second stage of the process already. See Section 9.

Palamedes. We implement our synthesis algorithm as a library called Palamedes for the Lean theorem prover; it builds Lean generators from predicates expressed as Lean functions. Working in a theorem prover brings us several benefits. First, the proofs produced by our deductive synthesis procedure can be checked by Lean's core checker; this ensures that, even if there were mistakes in the synthesis procedure, any successfully synthesized generator is guaranteed to produce exactly the desired set of values. Next, working in Lean means the implementation of the synthesis procedure can be simplified by using a popular proof search tactic, Aesop [31], 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. Our synthesizer is already quite powerful—it can, for example, synthesize generators for BSTs and well-typed simply-typed lambda calculus (STLC) programs—and the set of predicates it can handle can continue to grow modularly.

We evaluate Palamedes against two other methods for automatically obtaining PBT generators. Palamedes is considerably faster than the state-of-the-art approach to generator synthesis, Cobb [26], completing benchmarks like BST orders of magnitude faster. We also compare Palamedes to QuickChick [29, 39], which is able to automatically derive generators from inductive relations in Rocq; Palamedes is slower than QuickChick but imposes a lighter specification burden, and we discuss the trade-offs between the two approaches in detail.

Section 2 below outlines our synthesis procedure in the context of the BST example we have been discussing. The remainder of the paper presents the following contributions:

- We propose a system of deductive synthesis rules for correct generators (Section 3), proven correct relative to a denotational semantics of generators.

- We extend the resulting synthesis algorithm to work for generators of recursive data types (Section 4), using tools from the literature on recursion schemes to greatly simplify the synthesis process.
- We offer Palamedes, an implementation of our synthesis procedure embedded as a library in the Lean proof assistant (Section 5). This implementation strategy achieves performant synthesis that also maintains mechanized proofs of correctness, while borrowing key parts of the algorithm from Lean's existing infrastructure.
- We evaluate our approach on a suite of 32 benchmarks from the literature, demonstrating that our synthesizer is either faster or more flexible than state-of-the-art approaches (Section 6).

We conclude by discussing limitations (Section 7) and related and future work (Section 8 and Section 9).

2 Overview

In this section we illustrate the components of Palamedes with a running example: synthesizing a generator for binary search trees.

Synthesis for Simple Predicates. Our deductive synthesizer produces a PBT generator by recursive proof search. At every step, Palamedes attempts to match the current predicate (or goal) with the conclusion of an *inference rule* like the ones in Figure 3. Upon matching a rule, the process then continues with the premises of this rule. Concretely, to generate values satisfying a predicate P, we start with the statement

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

and successively refine the proof tree until we have a complete generator. For example, given a predicate like $\lambda\ a \Rightarrow a = 1 \lor a = 2$, the synthesizer would choose pick because the disjunction in its conclusion matches the form of the predicate. Its two component predicates, $\lambda\ a \Rightarrow a = 1$ and $\lambda\ a \Rightarrow a = 2$, become new goals for the synthesizer, both of which are matched by the rule for pure with no new further assumptions. The final result is generator pick (pure 1) (pure 2). We show a step-by-step derivation capturing this process in Appendix A.

$$\frac{}{\mathsf{pure}\ a' : \mathsf{Gen}\ (\lambda\ a \Rightarrow a = a')}$$

$$\frac{g_1 : \mathsf{Gen}\ P \qquad g_2 : \mathsf{Gen}\ Q}{\mathsf{pick}\ g_1\ g_2 : \mathsf{Gen}\ (\lambda\ a \Rightarrow P\ a \lor Q\ a)}$$

Fig. 3. Synthesis rules for pure and pick.

Matching inference rules to predicates is not always so straightforward. For example, in the process of synthesizing a generator for the `isBST` predicate in Figure 1 (in particular, in the sub-case generated by the `leaf` branch of the match), the synthesizer arrives at the goal predicate $\lambda\ a \Rightarrow \mathsf{true} \land a = \mathsf{leaf}$. This is logically equivalent to $\lambda\ a \Rightarrow a = \texttt{leaf}$, which matches the conclusion of the pure rule, but is not exactly the same. In a stand-alone synthesizer, we would need to add an inference rule for each such case, but in Palamedes we can use Lean's built-in theorems help us. In this case, Lean can automatically simplify the former predicate into the latter, allowing the synthesizer to apply the pure rule. The same concept means that Palamedes can apply the pick rule to choose between the `leaf` and `node` cases of the `isBST` predicate. Lean rewrites the **`match`** to a simple disjunction, which the synthesizer uses to produce the `pick` that appears in `genBST`.

$$\frac{g : \mathsf{Gen}\ P \qquad f : (a' : \alpha') \to \mathsf{Gen}\ (\lambda\ a \Rightarrow Q\ a'\ a)}{g \gg\!= f : \mathsf{Gen}\ (\lambda\ a \Rightarrow \exists\ (a' : \alpha'), P\ a' \land Q\ a'\ a)}$$

Fig. 4. Synthesis rule for bind ($\gg\!=$).

A hallmark of PBT generators is the ability to express dependencies between generated values. For example, `genBST` generates `x` by sampling a `choose lo hi` to pick a value between `lo` and `hi`, and it uses that x later in the generator. Our synthesizer handles these dependencies with the rule in Figure 4. The bind rule's conclusion matches a conjunction like the one in the body of `isBST`, where `lo <= x && x <= hi` is conjoined with `isBST l lo (x - 1) && isBST r (x + 1) hi`. When we apply the rule, we get two new obligations: one for a generator that can be sampled to give a value `x` (in `isBST` that's `choose lo hi`) and another for a generator that uses `x` to continue generating. (The binds that

produce values for `l` and `r` are a bit more complicated—we discuss them in a moment when we explain how we handle recursion.)

Recursive Predicates. Recursion is challenging for deductive synthesizers because generating recursive programs requires reasoning about termination. Previous work [23, 44] generates terminating recursive calls by proving that some measure decreases on each iteration, but generators often have much more complicated termination arguments than standard functional programs. Indeed, generators for some infinite sets (e.g., lists or trees) are often written in a way that is not guaranteed to terminate at all: they terminate with high probability but may have infinite execution traces.

```
def genBSTUnfold lo hi :=
  Tree.unfold
    (fun (lo, hi) ⇒
      if lo > hi then
        pure leafStep
      else
        pick
          (pure leafStep)
          (do
            let x <- choose lo hi
            pure (nodeStep (lo, x - 1) x (x + 1, hi))))
    (lo, hi)
```

```
def isBSTFold lo hi t :=
  Tree.fold
    (fun bl x br s ⇒
      match s with
      | (lo, hi) ⇒
        (lo <= x && x <= hi) &&
        bl (lo, x - 1) &&
        br (x + 1, hi))
    (fun _ ⇒ true)
    t
    (lo, hi)
```

Fig. 5. Left: a version of `genBST` using `Tree.unfold`. Right: a version of `isBST` using `Tree.fold`.

Luckily, it is possible to synthesize recursive generators without reasoning directly about individual recursive calls. The trick is to use a *recursion scheme*, and in particular an *unfold* (or anamorphism). Recursion schemes are popular in the functional programming literature [22, 59] as a way of abstracting common patterns of recursion, and it turns out that they work well for generators too. At a high level, an unfold describes a way to grow a data structure from a seed value. It takes as input a single step of the process, and then it "ties the knot" to turn that single step into a fully-fledged generator. You can see an example of this in the `genBSTUnfold` generator in Figure 5. The function passed to `Tree.unfold` either produces a `leafStep`, which says that the generator should produce a `leaf` and stop, or a `nodeStep`, which says that the generator should produce a `node` by continuing the process again with new seed values for the left and right sub-trees. This process ultimately does the same thing as `genBST` in Figure 2. We go into more detail on unfolds in Section 4; for now, it suffices to understand that unfolds let us synthesize generators for recursive generators without explicitly dealing with recursive functions.

How do we synthesize unfolds? By observing that unfolds are dual to another kind of recursion scheme, the *fold* (or catamorphism). Folds can express a wide range of recursive algorithms; in particular, folds can express predicates like `isBST` (`isBSTFold` in Figure 5). The fold version of the predicate essentially does the unfold in reverse: rather than take a seed value and expand it into a tree, it takes a tree and collapses it into a value. This is the duality that makes our approach work.

$$\frac{g : (b : \beta) \rightarrow \text{Gen}\ (\ldots)}{\text{Tree.unfold}\ g\ b : \text{Gen}\ (\lambda\, t \Rightarrow \text{Tree.fold}\ f\ z\ t = b)}$$

Fig. 6. Synthesis rule for Tree.unfold.

Putting this all together, we take a predicate like `isBST`, convert it to a form like `isBSTFold` that uses a fold, apply a rule like the one in Figure 6, and then continue with the deductive synthesis process outlined above to obtain a generator that looks like `genBSTUnfold`. This whole pipeline is automated

(see Section 4), and means that we can synthesize generators based on recursive predicates without actually needing our synthesis procedure to understand recursion explicitly.

As described above, this approach works for first- and second-order primitive recursive functions, i.e., those expressible with first- and second-order folds. Luckily, many common examples of predicates—including complex ones like filtering for well-typed programs—fit this description. In Section 4 we describe our handling of recursive predicates in more detail.

3 Deductive Synthesis for Generators

Palamedes proposes a novel approach to the constrained generator synthesis problem. This section introduces its representation of generators and some key definitions (Section 3.1), presents our core deductive synthesis rules for constructing generators (Section 3.2), introduces some handy base generators (Section 3.3), and describes an optimization procedure that can be applied to generators after synthesis (Section 3.4).

3.1 Generator Representation

Generators for values of type `a` are often represented as sampling functions of type `Seed→ a`, but we opt for a representation with a bit more flexibility. Taking inspiration from work on *free generators* [17], we represent a generator as an inductive datatype (Figure 7) that can be interpreted in multiple ways.

Each of these constructors has a standard interpretation as a procedure for sampling values. This interpretation is discussed in detail in Section 5; the intuition is as follows. The pure constructor represents a constant generator that always produces the same value. The $\gg=$ constructor represents sequencing of generators, sampling from one and passing the sampled value to a function producing another. The pick constructor represents a random choice between generators.

```
inductive Gen α where
  pure : α   → Gen α
  pick : Gen α → Gen α → Gen α
  bind : Gen α → (α → Gen β) → Gen β
  indexed : (ℕ → Gen (Option α)) → Gen α
  assume : (b : 𝔹) → (b = true → Gen α) → Gen α
```

Fig. 7. The datatype of generators. We use `>>=` or $\gg=$ as an infix variant of `bind`.

The indexed constructor represents an infinite family of generators, indexed by natural numbers; larger numbers give the generators more "fuel" to produce more values. (In lazy languages like Haskell this constructor is not necessary. But since we will be working in Lean, which is strict, we need it to be able to represent generators of infinite sets like natural numbers and lists.) The assume constructor represents a partial generator parameterized on a boolean condition; if true, it simply calls its argument; if false, it generates nothing. When interpreted as a sampling function, it is a partial function that can fail and need to be retried or backtracked around.

For proofs, we need an interpretation of generators that characterizes the *set* of values that they can produce via sampling. This interpretation is inspired by Paraskevopoulou et al. [40].

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

$$
\begin{aligned}
a \in [\![\text{pure}\ a']\!] &\iff a = a' \\
a \in [\![x \gg= f]\!] &\iff \exists\, a', a' \in [\![x]\!] \wedge a \in [\![f\ a']\!] \\
a \in [\![\text{pick}\ x\ y]\!] &\iff a \in [\![x]\!] \vee a \in [\![y]\!] \\
a \in [\![\text{indexed}\ f]\!] &\iff (\exists\, n, \text{some}\ a \in [\![f\ n]\!]) \\
&\qquad \wedge (\forall a\ n, \text{some}\ a \in [\![f\ n]\!] \rightarrow \text{some}\ a \in [\![f\ (n+1)]\!]) \\
a \in [\![\text{assume}\ b\ \text{in}\ x]\!] &\iff b = \text{true} \wedge a \in [\![x]\!]
\end{aligned}
$$

Note that the rule for indexed requires that the function passed to it be *monotonic*; without this condition, the generator could produce fewer values with more fuel, which would break completeness. Our synthesis procedure enforces this condition by construction. This setup means that support is not termination sensitive; a generator with support P will produce all values satisfying P (terminating when it does so).

```
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' >>= fun on' ⇒
              match on' with
              | none ⇒ pure none
              | some n' ⇒ pure (some (1 + n')))
  indexed go
```

Fig. 8. A generator of natural numbers.

Example: Generator for Natural Numbers. Figure 8 uses most of the above constructors to produce a generator for natural numbers. It uses indexed, since go is a fuel-indexed recursive function. If the fuel has run out, go fails with none. Otherwise, it makes a random choice between 0 and $1+n'$, where n' is generated by recursively calling go. The support of this generator is precisely $\mathbb{N}$.

Example: Backtracking Generator. While our ultimate goal is to avoid generators that search during generation, we still need such generators to be representable in our language. As we show in Section 3.4, our synthesis procedure works by first producing generators that need to backtrack and then attempting to optimize them. Here is an example of a generator that backtracks due to assume:

```
def genBacktrack := pick (pure 1) (assume false (fun _ ⇒ pure 2))
```

The support of this generator is $\{1\}$—i.e., when it generates a value, it always generates 1—but it will sometimes choose the right side of the pick and have to backtrack and try again.

Generator Correctness. We write $\text{Gen}_\alpha\ \varphi$ for the type of correct generators with respect to a predicate φ on a type α, eliding the latter if it is clear from context.

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

For example, the arbNat generator above can be given the type $\text{Gen}_{\mathbb{N}}\ (\lambda\, n \Rightarrow \top)$, while genBacktrack has type $\text{Gen}_{\mathbb{N}}\ (\lambda\, n \Rightarrow n = 1)$.

Assume-Freedom. Our gold standard for generators is exemplified by `genBST`—generators that behave 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, but there are situations in which it is not completely successful. In particular, it sometimes produces generators that use the assume constructor in ways that require backtracking. To distinguish these correct-but-suboptimal generators from the more performant ones we aim to synthesize as often as possible, we define:

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

3.2 Core Synthesis Algorithm

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

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

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

$$\frac{(x : \mathsf{Gen}_\alpha\ P) \in \Gamma}{\Gamma \vdash x : \mathsf{Gen}_\alpha\ P}\ \text{S-Assumption} \qquad \frac{b{:}\beta, \Gamma \vdash g : \mathsf{Gen}_\alpha\ P}{\Gamma \vdash (\lambda\, b \Rightarrow g) : (b : \beta) \rightarrow \mathsf{Gen}_\alpha\ P}\ \text{S-Intro}$$

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

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

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

Fig. 9. Palamedes core synthesis rules.

Pure and Pick. We already saw the first two synthesis rules, S-Pure and S-Pick, in Figure 3. The former says that we can synthesize a value that is equal to a constant using pure, and the latter says that we can synthesize for a disjunction using pick.

Convert. We also saw in Section 2 that synthesis rules may not apply directly to the goal as stated. For example, the predicate $\lambda\ a \Rightarrow \mathsf{true} \wedge a = \mathsf{leaf}$ *almost* matches the S-Pick rule, but not quite. In these situations, we rely on Lean's simplifier to convert the goal into a form that matches one of our synthesis rules. Formally, this process is done via the S-Convert rule in Figure 9.

Assumptions and Functions. If the generator we need is already available in the typing context Γ, we can just use it via the S-Assumption rule. When the synthesis goal is a function returning a generator, we can apply an introduction rule. The S-Intro 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). The S-Uncurry rule says that we are free to synthesize a curried function and then uncurry it, if the goal is to produce a function with a tuple argument.

Bind. We can use $\gg\!=$ to chain generators together, as demonstrated in Section 2:

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

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

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

Case Splitting. If a generator needs to do different things based on the value of a variable in the context, it can use rules like S-SPLITBOOL or S-SPLITNAT to introduce pattern matches. Rules for inductive types other than booleans and numbers can be derived from their definitions (see Section 5).

3.3 Standard Library Generators

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

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

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

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

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

Fig. 10. PALAMEDES standard library synthesis rules.

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

```
def choose (lo hi : ℕ) := if lo = hi then pure lo else pick (pure lo) (choose (lo + 1) hi)
```

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

The corresponding synthesis rule, S-CHOOSE, appears in Figure 10.

In a synthesis context, it may not always be easy to show that $lo \leq hi$—it may not even be true. This motivates a second way to synthesize choose that checks its precondition dynamically, the S-CHOOSEPARTIAL rule. Both rules are valid, but the second introduces the potential for the generator to fail: if it is called with $lo > hi$, it will not be able to produce a value. This generator is sound in the sense that, if it produces a value, then that value satisfies the given condition—but it is not ideal for testing. Luckily, we can usually optimize the assume away later (see below).

Elements. The elements generator picks a random value from a list:

```
def elements xs := match xs with | [x] ⇒ pure x | x :: xs' ⇒ pick (pure x) (elements xs')
```

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

Greater Than and Less Than. Finally, there are greaterThan and lessThan generators that take a single natural number and can generate any number respectively greater or less than that number. They are similar to choose, so we defer them to Appendix B.

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:

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

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

Fig. 11. Optimization rules for generators. Rules (1) and (2) are standard monad equivalences, (3) and (4) describe how assumes interact with binds, and (5) and (6) actually lift assumes out of choices.

Logically, this assume is not necessary. As written, the generator makes a choice and then fails if it happened to choose the right branch and $lo > hi$. But it could just as well check $lo \leq hi$ first and only choose the right branch if the check succeeds. Generalizing this observation, we design a set of *optimization rules* that rewrite generators to avoid failures; these appear in Figure 11. The rule we need for the above case is rule (6), which rewrites a pick containing an assume to an if statement that checks the assumption before the choice.

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

These rules are not complete: they may still leave assumes in the generator. For example, a pick with assumes on both sides will still fail if both conditions are false. In some of these cases, a stronger optimizer may be able remove more assumes, but most assumes left after optimization indicate fundamental backtracking in the algorithm that the synthesis process found. Happily, for the vast majority of the examples in Section 6, they can derive a generator that is assume-free.

This optimization procedure is critical for some of our most interesting examples, including the BST example that we introduced in Section 2. When synthesizing a generator for BSTs, Palamedes uses S-ChoosePartial to pick a value for a node that is within the correct range, and then later the optimizer lifts the bounds check out of the assume and into a less expensive check. This is why the generator that we synthesize for BST (roughly `genBSTUnfold` in Figure 5) checks `lo > hi`.

4 Synthesizing Generators for Recursive Predicates

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

At a high level, the approach in this section takes a recursive predicate, normalizes it, and then applies a synthesis rule like the ones from the previous section. The pipeline that we implement is shown in Figure 12. The key takeaway is that, rather than give the synthesizer direct access to indexed and recursion, we synthesize recursive generators through higher-level rules. This approach does have limitations (e.g., it cannot create generators like elements that iterate over one structure and produce another), but it is highly effective for many predicates over data structures.

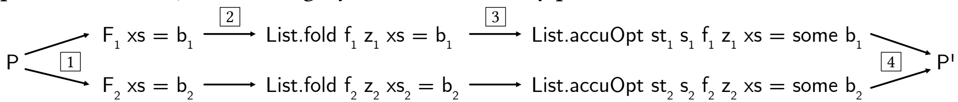

Fig. 12. How a list predicate is normalized for synthesis.

4.1 Background: Recursion Schemes

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

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

```
def List.fold
    (f : α → β → β) (z : β)
    (xs : List α) : β :=
  match xs with
  | [] ⇒ z
  | x :: xs' ⇒ f x (fold f z xs')

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 (accu st (st x s) f z xs')
```

Fig. 13. Definitions of folds and accumulators.

Folds. The simplest recursion scheme is a *fold* or *catamorphism*. Figure 13 shows an implementation of fold for the List datatype. This function takes as arguments a “base-case” z and a “step function” f. We call the type β the “collector” for the fold.[1] When the list is empty, we return z. When the list is a cons cell, we recursively call List.fold f z on its tail and use f to combine the resulting value with the value at the head.

Note that information here flows backward, from the tail of the list to the head.[2] We first compute something about the tail, without considering the value at the head, and only at the end do we actually put the two together. This point will be useful to remember.

```
def isSorted xs :=
  List.accu
    (fun x _ ⇒ x)
    0
    (fun x b lo ⇒ lo <= x && b)
    (fun _ ⇒ true)
    xs
```

Fig. 14. Checking if a list is sorted with List.accu.

More Advanced Recursion Schemes. While List.fold can represent all primitive-recursive functions over lists [22], we will see that it is not always ergonomic to use it. For this reason, researchers have identified specialized recursion schemes, each capturing some common pattern of recursion [59]. Of note here is the *accumulation* pattern [41], which we present with some minor simplifications in Figure 13. Accumulations are similar to folds, but they pass information in both directions. The collector value of type β is still passed backwards through the list, but we add a new type parameter σ representing an “accumulation state” that flows forward. The accumulation takes an initial state s as input, along with a “state update function” st that says how the state changes.

We can use List.accu to naturally implement some functions that would be awkward with List.fold. For example, the function in Figure 14 checks if a list of natural numbers is sorted. The accumulation state is the minimum value allowed in the remainder of the list; it is initialized to 0 and updated to the most recently seen value at each step. The step function then checks that $lo \leq x$ and conjoins that with the boolean computed from the tail of the list, to ensure that the tail of the list is also sorted. We could implement this same function with List.fold, but this would require the collector to be a higher-order function, which would be difficult for the synthesizer to deal with.

[1] Others call this value the “accumulator,” but we use “accumulation” to refer to a type of fold [41] and want to avoid confusion.
[2] I.e., this is a “right fold” over the list (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.

```
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
```

Fig. 15. Definitions of *optional* folds and accumulators.

Optional Folds. We also need versions of List.fold and List.accu that capture *optional computations* (Figure 15). We discuss use cases for these in the next section; for now, notice that they behave the same as List.fold and List.accu, except that, if any step evaluates to none, then the whole thing does.

Unfolds. The final recursion scheme we examine in detail is the *unfold* or *anamorphism*. Unfolds are the inverse of folds: whereas folds collapse data structures into compact values, unfolds expand values into data structures. The unfold for lists is shown in Figure 16.

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

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

Fig. 16. Definition of unfold for lists.

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

4.2 Generators for Inductive Data Types

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

As a first example, given a predicate that uses List.fold to check that a list has a given length, we can use List.unfold to write a generator for values satisfying this predicate (Figure 17). At each unfolding step, it checks the seed value n. If $n = 0$, it generates none, indicating that the list

```
def isLengthK (k : ℕ) (xs : List ℕ) :=
  List.fold (fun _ b ⇒ 1 + b) 0 xs = k
def genLengthK (k : ℕ) : Gen (List ℕ) :=
  List.unfold
    (fun n ⇒
      match k with
      | 0 ⇒ pure none
      | 1 + k' ⇒
        arbNat >>= fun x ⇒
        pure (some (x, k'))) n
```

Fig. 17. A predicate and generator for lists of length k.

should end (since n is the target length of the list). Otherwise, it generates an arbitrary natural number x and yields some $(x,n-1)$ to indicate that the list should continue with a cons cell containing x, plus a decremented target length.

How might we derive genLengthK from isLengthK? The key observation is that isLengthK and genLengthK have an inverse relationship—whenever genLengthK takes a step, it is guaranteed that isLengthK can undo that step.

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

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

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

The fold and unfold are inverses here because, for each step the unfold takes, the fold is guaranteed to be able to "fold that step back up" and recover the seed. Another perspective comes from the observation that a fold passes information backwards in a list from the tail to the head; the unfold does the opposite, passing information forwards in such a way that the fold would always compute the same thing going the other way.

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

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

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

Our system can use this rule to synthesize genLengthK from the definition of isLengthK.

4.3 Handling More Complex Folds

The S-List-Unfold' rule is key to understanding the basics of our approach, but it is not quite powerful enough for our most interesting use-cases. For example, consider a predicate that checks that all elements of a list are equal to 2:

```
def isAllTwo (xs : List ℕ) := List.fold (fun x b ⇒ x = 2 && b) true xs = true
```

Using S-List-Unfold', we could turn this directly into a generator that looks like this:

```
List.unfold (fun b ⇒ if b then pick (pure none) (pure (some (2, true))) else ...)
```

The true branch here is exactly what we want; it chooses between `none`, which stops generation, or `some (2, true)` which says to put a value `2` at the head of the list and continue. But false branch is unnecessary; it will never be executed, since b starts as true and remains true every step through the unfold. We would prefer to avoid synthesizing the false branch at all.

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

```
List.foldOpt (fun x () ⇒ if x = 2 then some () else none) (some ()) xs = some ()
```

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

As we saw earlier, List.foldOpt allows the fold to pass information backward from the tail of the list to the front, but it does not allow state to flow forward. We can be one step more general and consider predicates expressed with List.accuOpt. This leads to our most general synthesis rule for

recursive predicates: the S-List-Unfold rule, which subsumes and replaces S-List-Unfold':

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

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

This rule is not pretty, but its basic operation is exactly the same as S-List-Unfold'. It just allows for more flexible handling of hidden invariants (see isAllTwo) and more interesting state passing.

If we rewrite isSorted from the beginning of this section using List.accuOpt instead of List.accu, we can use S-List-Unfold to derive the generator in Figure 18, which behaves the same as one an expert user might write. At each step, it either ends the list or generates a new value x that is greater than or equal to lo, puts it in the list, and continues with $lo = x$.

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

Fig. 18. A generator of sorted lists.

4.4 Tupling Predicates

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

```
def isAllTwoLengthK (k : ℕ) (xs : List ℕ) := isAllTwo xs && isLengthK k xs
```

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

We have two options for handling such situations. The first is to draw from the literature on program calculation and apply a *tupling* transformation [5, 42] to combine the conjuncts. These transformations were originally designed for optimizing functional programs, but they are a natural fit for this problem. Another is to adapt the "merging" procedure for inductive relations described by Prinz and Lampropoulos [45]. Indeed it turns out that, in the case of predicates written with List.accuOpt, these concepts coincide!

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

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

4.5 Putting it All Together

The machinery described in the previous subsections is assembled as follows into a standardized workflow for transforming predicates on recursive data structures into a normal form (see Figure 12).

(1) Normalize the predicate to the form $\lambda\ xs \Rightarrow F_1\ xs = b_1 \land \cdots \land F_n\ xs = b_n$.

(2) Rewrite each F as a fold. Concretely, search for z and f satisfying the equations $F\ [\,] = z$ and $F\ (x :: xs) = f\ x\ (F\ xs)$. If these equations are satisfied, then F can be rewritten as List.fold f z. This is sometimes referred to as the "universal property of fold" [22, 33].
(3) Rewrite each fold as an optional accumulation based on its return type. For example, isAllTwo, which collects an always-true boolean, would be transformed differently from isSorted, which collects a higher-order function.
(4) Tuple the n different branches of the predicate together.

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

Everything in this section has been phrased in terms of the List data type, but there is actually nothing in the workflow that is specific to lists. Any recursive data type composed of products and sums admits operations analogous to List.fold, List.accuOpt, etc. This means that the pipeline from Figure 12 directly generalizes to a wide range of recursive data structures.

The case studies in Section 6 required implementing this pipeline (plus other utilities, e.g., for case splitting) for five data structures: lists, binary trees, STLC types, STLC terms, and stacks (from [21]). The implementation process is quite mechanical, as the definitions follow the structure of the data type and its constructors, and it should be straightforward to automate it, either via meta-programming or using "quotients of polynomial functors" (QPFs) [4]. We leave this engineering for future work.

5 Palamedes: Synthesizing Generators in Lean

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

5.1 Overview

Palamedes is packaged as a library for the Lean theorem prover. Using Lean's powerful meta-programming capabilities, everything can be implemented as standard Lean code, and using it requires only importing the library. Then, starting with the BST validity predicate from Figure 1, a user can invoke Palamedes's `generator_search` tactic to synthesize a generator for BSTs:

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

When the Lean compiler builds the file, it runs `generator_search` to synthesize an appropriate generator—in this case, `genBST` from Figure 5.

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

5.2 The Synthesis Algorithm

We can think of "`generator_search P`" as calling three tactics in order: `synthesize`, `optimize`, and `prove_assume_free`. We now describe each of these steps in more detail.

Step 1: Synthesize. The `synthesize` tactic solves a goal of type `CorrectGen P` by applying the rules in Section 3.2. The procedure uses Aesop, a tactic in Lean that performs best-first proof search [31]. Aesop takes a large set 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. Aesop has a timeout, which the user can change. Relying on Aesop in this way turns out to be extremely effective in terms of both the results for our case studies and ease of implementation. The rules we provide to Aesop mirror the ones described in Section 3; they are given in full in Appendix C.

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

Step 3: (Optionally) Prove Assume-Free. After optimization, we use straightforward proof automation to check whether the synthesized generator makes nontrivial use of the assume constructor. All but three generators we synthesize in our evaluation are assume-free, and the automation easily proves that fact; we emit a warning if the assume-free check fails. We say that this step is optional because Palamedes does not categorically reject generators that contain assumes. For example, two benchmarks that we borrow from Prinz and Lampropoulos [45], AVL trees and red-black trees, make nontrivial use of the assume constructor. We discuss this more in Section 6.3.

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 just want to keep the codebase simple, 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; otherwise it will re-synthesize the generator when reloading the file. Users may also want to *render* the synthesized generator as a concrete program that they can edit. They might, for example, want to add weights to bias the distribution or add size bounds to ensure generated values do not get too big. (This may make the generator incomplete, so we do not do it during synthesis, but an expert user may decide the incompleteness is worth it on a case-by-case basis.) The `generator_search?` tactic invokes the same synthesis procedure and then provides the user with a "try this" widget in their editor; clicking pastes the text of the generator into their file.

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

6 Evaluation

To evaluate Palamedes, we first examine its performance, comparing it with two state of the art competitors. We also qualitatively compare our generators to handwritten ones from the literature. We address the following research questions:

RQ1 How does Palamedes compare to Cobb, a recent approach that uses program synthesis to repair incomplete PBT generators?

RQ2 How does Palamedes compare to QuickChick, the standard tool for deriving generators from inductively defined specifications in Rocq?

RQ3 How do the generators synthesized by Palamedes compare to ones that expert users write?

Benchmarks. We use several sets of benchmark to answer these questions:

- Main: A set of 32 predicates demonstrating specific aspects of Palamedes's synthesis algorithm, including low-level examples from the text above and others drawn from the PBT literature [26, 29, 45]. A detailed table of the benchmarks is in Appendix E.
- Cobb: To compare to the state of the art in RQ1, we used versions of the benchmarks in Main in the input format for Cobb [26], a synthesis-guided repair tool for generators. For benchmarks that were not drawn from Cobb's evaluation, we needed to create sketches of generators to repair.

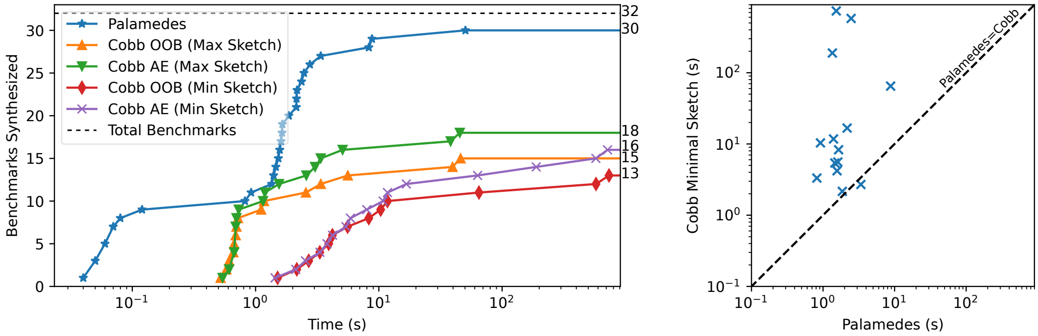

Fig. 19. Comparing Palamedes and Cobb. Left: Number of generators synthesized by Palamedes and Cobb (OOB = "Out of the Box", AE = "Additional Engineering"). Right: Per-benchmark comparison for the 16 benchmarks synthesized with both sketches (above the line means Palamedes is faster). Time axes are logarithmic.

Following Cobb's existing benchmarks, we turned each target generator into two sketches: a *minimal sketch* that left only the control flow of the generator with holes, and a *maximal sketch* that only removed one element from the generator. (Of the possible elements, we removed the one that allowed Cobb to perform best.)

- Inductive: To test RQ2, we used versions of the same 32 benchmarks defined as Rocq inductive predicates, so that they are a fit for QuickChick [29]. Many of the benchmarks (such as the tree- and STLC-based ones) already had inductive variants, as they were themselves originally adapted from QuickChick's benchmark suite [26]; the rest were straightforward to translate manually.

For RQ3, we implemented a selection of generators from Main by hand, and below we qualitatively compare those generators with the corresponding synthesized ones.

Experimental setup. All experiments were performed on an M1 MacBook Pro with 8 cores and 16 GB of memory using Lean v4.21.0, with a timeout of 900 seconds. Time measurements of Palamedes are averaged over 30 runs and measured in seconds. Cobb was run from a local installation build following the instructions provided by the Cobb artifact [27]. The internal cost used by Cobb in lieu of a timeout was set to 50,000,000, cancelling out cases where Cobb ran out of "cost" before our timeout.

6.1 RQ1: Comparison with Cobb

We ran each of the Main benchmarks in Palamedes and their Cobb counterparts in Cobb and measured the time until a generator was found by each system. The results are shown in Figure 19. Synthesis times for Palamedes range from around 40ms to 50s on `isRBT`, which has multiple nested case splits. Palamedes finds generators for 80% of the Main benchmarks in under 3 seconds.

The two benchmarks that Palamedes fails to solve are `isUnique` and `hasDuplicates` from Cobb's benchmark set. These examples have a large gap between their recursive forms and their accuOpt form, and Palamedes currently cannot automate that conversion—Aesop fails to make progress and gives up in under 2s for both. Cobb has the benefit of a sketch to help scaffold these tricky cases.

From maximal sketches (the generator with one hole) Cobb's fastest benchmark, `isSortedBetween`, takes just over half a second. With minimal sketches, i.e., only the generator's control flow, Cobb solves its fastest benchmark, `isUnique`, in just over 1.5 seconds, and it takes 12 minutes to synthesize a generator for `isBST`. In total, Cobb solves 13 of the 32 benchmarks with a minimal sketch and 15 with a maximal sketch out of the box. Our team spent a collective 20 person-hours attempting to adapt Cobb to the 19 missing benchmarks, and we were able to get another 3 working with both maximal and minimal sketches. Of the other 16, 9 were very simple (e.g., $\mathbf{v} = 2$); Cobb outputs "nothing to repair," even though the generator is incomplete. We are certain Cobb can handle these in theory;

that was likely a bug in our setup or the implementation. The other 7 were more complicated (e.g., the Palamedes version of `isRBT`, which is stronger than the one in the Cobb paper). One of these (`isTrue`) cannot be encoded because the current implementation only supports integer lists; the rest might be implementable with more effort.

Cobb was faster than Palamedes on one benchmark—`isLengthKAllTwos`—by 680ms (25%). Here, Palamedes needs to find the case split on `length`, whereas Cobb gets this decision as input in the provided sketch.

Overall, Palamedes outperforms Cobb while requiring less information to produce a generator: only the predicate, with no additional sketch. We find that Palamedes **is more expressive than Cobb and finds generators faster, despite its search needing to solve a harder problem.**

6.2 RQ2: Comparison with QuickChick

Palamedes and Cobb use similar techniques to solve similar problems. By contrast, a direct comparison to the next-closest state-of-the-art tool, QuickChick, is harder. QuickChick does not search for generators; instead it works more like a compiler, producing a generator almost instantly by walking over a predicate in a single pass. The trade-off is that QuickChick's predicates must be expressed as (slightly restricted) inductive datatypes in Rocq, rather than as functions.

To examine the relationship between the structure of the predicate and the quality of the derived generator, we ported the Main benchmarks to QuickChick and characterized the effort needed to achieve non-backtracking generators. The results appear in detail in Appendix E.3; we summarize and discuss them here.

All 32 of the benchmarks needed to be in inductive form: for some, inductive versions were already present in QuickChick's examples as they were adapted from there; the rest were converted manually. Of the latter, one ($\exists a,\ a = 3 \vee \mathbf{v} = a + 1$) could not be straightforwardly ported as QuickChick does not handle existential quantifiers. To effectively handle conjunctions of predicates constraining the same variable, QuickChick requires users to manually *merge* the conjoined predicates [45]. Ten of the benchmarks, e.g., `AVL`, required manually invoking the merging procedure. Finally, one benchmark, STLC's `isWellTyped`, also required a specific constraint ordering in its definition for effective generation, which is not a concern for Palamedes. Moreover, any predicates expressed naturally in a higher-order style (e.g. using folds) would need to be manually translated to variants using direct recursion.

In general, QuickChick's automation works extremely quickly when it works, but it may require some careful setup from the user (including translation to an inductive data type) before everything falls into place. By contrast, Palamedes can search for solutions that do not follow the precise structure of the predicate. This requires less user setup, especially when predicates are defined as functions, but it is slower and less predictable. Ultimately, we find that Palamedes **is a useful complement to QuickChick, applying more directly to the common case where predicates are expressed as functions, but with trade-offs in terms of performance.**

6.3 RQ3: Synthesized vs. Handwritten Generators

Finally, we explore how the generators that Palamedes produces compare to ones that expert users might write by hand.

First, we consider our synthesized generator for well-typed STLC terms,

shown in full in Appendix G. The generator we synthesize naturally implements a method for generating well-typed terms popularized by prior work [38], in which the generator first generates a random type and then generates a random term of that type.

For comparison, we provide a handwritten STLC generator that the authors wrote in Appendix F. The handwritten generator reduces a bit of code repetition and eta-reduces a match expression, but

the asymptotic performance of the generators would be the same—neither would backtrack, and both would contain the same basic clauses. Fundamentally, both of these generators implement the same generation algorithm with the same time complexity.

Next, we consider our synthesized generator for AVL trees. Palamedes's AVL tree generator appears in Appendix H. As we mentioned briefly in Section 5.2, this generator is one of the few that we cannot remove all assumes from. The generator we synthesize is still quite efficient, and the AVL tree generator that Prinz and Lampropoulos [45] present in their paper on merging inductive relations also backtracks in the same way, but nevertheless our generator is not perfect.

There are two common ways to generate AVL trees without any backtracking, which we show in Appendix I. The first is a particularly high-effort option that requires careful insight and analysis to get right. In practice, it is uncommon for real users to go to this effort. More common is an approach that generates an AVL tree by first generating a list of values and then `insert`ing those values one by one into an empty tree. We could imagine Palamedes looking for a generator like that someday, since it is technically possible to express with our generator language, but we leave this to future work.

Appendix J shows four more generators that Palamedes synthesized, side-by-side with ones that we wrote manually. For all of these, there were no functional differences between the generators we wrote and the synthesized ones.

To summarize, we find that Palamedes **often implements the same fundamental generation algorithm as handwritten versions (i.e., with the same time complexity), but this does not necessarily mean that synthesized generators are as fast to run (e.g., due to synthesis artifacts or tuning issues).**

7 Limitations

Our approach has some limitations. First, as with any synthesis algorithm, Palamedes is necessarily partial; there are many Lean predicates for which Palamedes does not successfully find a generator. (We discussed some of these in Section 6.1.) A core reason for this is our handling of recursive predicates; by handling only predicates that can be expressed in terms of `accuOpt`, we limit ourselves to only first- and second-order primitive recursive programs. Synthesis may also fail due to missing base generators: for example, uses of assume need to be explicitly included for synthesis (e.g., in the definitions of S-Choose and S-Elem), so generators that require backtracking (e.g., the AVL example) may require manual extension of the library. Luckily, Palamedes is indeed extensible—users can add new proof automation, library generators, and synthesis rules—so its partiality can be mitigated over time. In Section 9, we discuss plans to extend the set of predicates that Palamedes can handle.

Second, as we mention in Section 1, this paper works with generators as nondeterministic programs, rather than as true probability distributions. We discuss how we hope to *tune* our synthesized generators, optimizing their distributions as well, in Section 9.

Finally, Palamedes does not incorporate *sizes*, a classic PBT generator feature. QuickCheck generators [10] 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. Our approach is compatible with sized generation—prior work [40] gives a semantics for internal sizing that plays well with our definitions—but we do not yet know how the synthesizer should use sizes. Currently, then, there are two ways to control the sizes of generated values in Palamedes: (1) Users can augment their specifications with explicit size constraints. E.g., `isLengthKAllTwos` has an argument `k` to control the length of the generated list; the tupling transformation means this can be done by simply conjoining an extra predicate. (2) Users can manually modify their generators after synthesis to add size control.

8 Related Work

The Constrained Random Generation Problem. As discussed in Section 1, many approaches to the constrained random generation problem for testing have been proposed over the years. In rough order of publication: Dewey et al. [12] propose input generation via constrained logic programming (CLP). This can handle some realistic examples like "red-black trees that are guaranteed to rebalance when a particular input is inserted" that would likely require some extra infrastructure in Palamedes. Seidel et al. [49] search for valid inputs via SMT in a refinement type system; their technique asks a solver for an example of a valid input, then an example of a valid input that isn't the first they were given, and so on. Claessen et al. [9] use laziness in Haskell to find valid inputs, filtering bad choices as early as possible provided that the precondition is can be executed sufficiently lazily. Reddy et al. [47] guide generation via reinforcement learning, maximizing the chance that the next generated input is both valid and different from past inputs. Goldstein and Pierce [17] rely on Brzozowski derivatives to focus in on particular parts of the input space that might be more dense with valid values. Steinhöfel and Zeller [50] again use SMT for generation, but this time in a stand-alone system that layers the solver's feedback on top of grammar-based generation. Xia et al. [55] call a large language model to search for inputs that satisfy a function's preconditions.

These related approaches are all exciting and useful in various cases, but they solve a fundamentally different problem from the one solved by PALAMEDES: they actively guide generation towards valid values *during testing*, rather than searching for a correct generator ahead of time. In particular, all of these algorithms do potentially exponential work when generating each input, while an assume-free generator produced by Palamedes often does work linear in the size of the value being generated. In cases where a generator is written once and then run many times, which is common in practice [16], paying the cost of search up-front is a good investment.

The Constrained Generator Synthesis Problem. As we discuss in detail in Section 6, both QuickChick [29, 39] and Cobb [26] address the constrained generator synthesis problem in as we define it.

There are two other related tools that we did not get a chance to evaluate against, since both are predecessors of and inspiration for the generator automation in QuickChick. First, Luck [28] is a standalone language for writing predicates (with some light annotations) from which reasonably efficient generators could be derived. Luck's predicate language is almost completely subsumed by QuickChick's handling of inductive relations. Second, before the work in Rocq there was work on deriving generators in Isabelle [7, 8]. The work in Isabelle is different from QuickChick in a few ways, and Isabelle can handle some predicates expressed as functions (rather than inductive relations). That said, the approach still fundamentally follows the structure of the predicate to derive a generator, rather than implementing a more flexible search like PALAMEDES does. We would like to compare our approach to both Luck and Isabelle in the future.

Deductive Program Synthesis. Deductive program synthesis takes a specification in some logic and searches for a proof that the specification holds; proofs found by the search can be read as correct-by-construction programs. The simplest version of deductive synthesis is *type-directed synthesis* [15, 19, 36, 37], where proofs are by application of typing rules. This has been extended to expressive type systems like refinement types [43] and semantic types [18]. (Cobb [26] also performs type-directed synthesis to generate repairs, but uses a *bottom-up* approach instead of a deductive one.) Several systems extend this approach to other logics [11, 25, 46], including separation logic [13, 23, 44].

PALAMEDES builds on the extensive body of work on deductive synthesis, particularly on SuSLik [44] and its followup work [13, 23, 53]. SuSLik synthesizes imperative programs by iteratively applying a proof rule for the next statement and leaving the remaining obligation to specify the rest of the program; we take inspiration from this process in how PALAMEDES handles binds. Our use of anamorphisms (unfolds) to build generators is closely related to Hong and Aiken [20]'s use of

paramorphisms for synthesizing recursive algorithms. However, Palamedes's target domain has not been explored by prior work, and its implementation differs in key ways. Unlike prior work, Palamedes relies on Lean for many aspects of proof search; e.g., avoiding explicit reasoning about termination and solving auxiliary lemmas with Aesop [31]. And, excepting Fiat [11] and SuSLik [53], prior work does not produce mechanized correctness proofs. Palamedes's deduction rules are proved correct in the Lean theorem prover, and those proofs are combined to prove the final generator is correct, strengthening the guarantee that resulting programs are correct by construction.

9 Conclusion and Future Work

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

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

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

which (assuming `pick` chooses branches with equal probability) will produce `1` with probability 0.5, `2` with probability 0.25, and `3` and `4` with probability 0.125. This bias towards `1` is probably not something the user wants.

Luckily, there are ways to address this problem. As a simple solution, we could implement an optimization pass that re-associates right-heavy nested sequences of `pick`s to prefer more balanced trees. Less naïvely, recent work has shown that probabilistic programming languages like Loaded Dice can be used to automatically tune generators to user-specified distributions [52]. We plan to implement a translation from our generators into Loaded Dice, giving users comprehensive control over generator distributions.

Correct Generators for Everyday Developers. Implementing Palamedes as a Lean library has myriad benefits, but it has one major downside: theorem provers are relatively inaccessible to everyday software developers, so the tool in its current form is unlikely to see broad adoption. However, we see a clear path towards impact in the software engineering industry, by using Palamedes as a back end for user-facing tools. As a first step in this direction, we plan to (1) embed a subset of Python's semantics in Lean, (2) compile Python predicates to that sub-language, (3) synthesize generators for those predicates, and (4) render the synthesized generators as Hypothesis [32] strategies. If successful, we hope to push this paradigm even further, using Palamedes as a backend for synthesizing generators for other languages and PBT frameworks.

Better Automation and Algorithmic Improvements. As we demonstrated in Section 6, Palamedes is already flexible enough to synthesize a wide range of generators, but the algorithm may be further improved by ongoing enhancements to Lean's proof automation. For example, Lean now has tactics for automating proof search with SMT solvers [35] and e-graph rewriting [48]. LLM-based proof automation is also an active area of study [14, 24, 30, 51, 57, 58], which may unlock more flexible approaches to proof generation.

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

Acknowledgements

The work by Harrison Goldstein was funded by the Victor Basili Postdoctoral Fellowship from the University of Maryland.

The work by Hila Peleg was funded by the European Union (ERC, EXPLOSYN, 101117232). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

The work by Cassia Torczon, Daniel Sainati, Leonidas Lampropoulos, and Benjamin C. Pierce was funded by a number NSF grants and fellowships: *SHF: Medium: Usable Property-Based Testing, NSF #2402449*; *CISE: Graduate Fellowships Grant, NSF #2313998*; and *CAREER: Fuzzing Formal Specifications, NSF #2145649*. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We also thank Mike Dodds, Shachar Itzhaky, Nadia Polikarpova, and Ilya Sergey for their feedback and input at various stages of this project.

Data Availability Statement

The code artifact for this paper is available on Zenodo at https://doi.org/10.5281/zenodo.19073205. We are also developing Palamedes as an open source project at https://github.com/hgoldstein95/palamedes-lean (at the time of writing, it is still under active development).

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] 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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] Kyle Dewey, Lawton Nichols, and Ben Hardekopf. 2015. Automated Data Structure Generation: Refuting Common Wisdom. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 32–43. https://doi.org/10.1109/ICSE.2015.26 ISSN: 1558-1225.

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

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

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

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

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

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

[19] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2491956.2462192

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

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

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

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

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

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

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

[27] 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/doi:10.5281/zenodo.16599071

[28] 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.

[29] 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.

[30] 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].

[31] 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

[32] 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.

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

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

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

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

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

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

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

[40] Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015. Foundational Property-Based Testing. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 325–343. https://doi.org/10.1007/978-3-319-22102-1_22

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

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

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

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

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

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

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

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

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

[50] 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] 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].

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

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

[54] 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

[55] 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].

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

[57] 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].

[58] 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].

[59] 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].

[60] Francille Zhuang. 2026. Ariadne: Automatically Tuning Generator Weights Using Dynamic Sampling. Poster presented ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).

A Derivation of a Simple Generator

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

apply S-Pick

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

apply S-Pure **on the left**

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

apply S-Pure **on the right**

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

B Greater Than and Less Than

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

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

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

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

C Synthesis Rules for Aesop

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

Rule	Precedence
uncurry_intro	100%
apply s_arbitrary_<T>	100%
assumption	99%
apply convert (**by** match_pure) s_pure	99%
apply convert (**by** match_pick) (s_pick _ _)	99%
apply convert (**by** match_bind 0) (s_bind _ _)	99%
apply convert (**by** match_bind 1) (s_bind _ _)	99%
apply convert (**by** match_greaterThan) s_greaterThan	99%
apply convert (**by** match_lessThan) s_lessThan_partial	99%
apply convert (**by** match_between) (s_between (**by** solve_between))	99%
apply convert (**by** match_between) s_between_partial	99%
apply convert (**by** match_elements) s_elements_partial	99%
apply convert (**by** match_<T>_unfold) (<T>.s_unfold _)	99%
split_cases 0 <T>.split	5%
split_cases 1 <T>.split	5%
split_cases 2 Nat.split	5%
split_cases 3 Nat.split	5%

D Predicate Definitions

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

```
    (f_abs : Ty -> α -> α)
    (f_app: α -> α -> α)
    (t : Term) :
    α :=
  match t with
  | .unit ⇒ z
  | .var n ⇒ zn n
  | .abs τ t' ⇒ f_abs τ (Term.fold z zn f_abs f_app t')
  | .app t1 t2 ⇒
    f_app (Term.fold z zn f_abs f_app t1) (Term.fold z zn f_abs f_app t2)
def isAllTwos : List Nat -> Bool
  | [] ⇒ true
  | x :: xs ⇒ x = 2 && isAllTwos xs
def isEvenLen : List α -> Bool
  | [] ⇒ true
  | _ :: xs ⇒ !(isEvenLen xs)
def isAllTwosEvenLen (xs : List Nat) : Bool :=
  isAllTwos xs && isEvenLen xs
def isIncreasingByOneAux (xs : List Nat) (prev : Nat) : Bool :=
  match xs with
  | [] ⇒ true
  | x :: xs' ⇒ x == prev + 1 && isIncreasingByOneAux xs' x
def isIncreasingByOne (xs : List Nat) : Bool :=
  isIncreasingByOneAux xs 0
def isLengthKAllTwos (k : Nat) (xs : List Nat) : Bool :=
  xs.length == k && isAllTwos xs
def isSortedBetween : List Nat -> Nat × Nat -> Bool := fun xs (lo, hi) ⇒
  match xs with
  | [] ⇒ true
  | x :: xs' ⇒ (lo <= x && x <= hi) && isSortedBetween xs' (x, hi)
def isTrue : List α -> Bool
  | [] ⇒ true
  | x :: xs ⇒ (fun _ ⇒ true) x && isTrue xs
def hasDuplicatesAux (xs : List α) (soFar : List α) :=
  match xs with
  | [] ⇒ false
  | x :: xs' ⇒ List.elem x soFar || hasDuplicatesAux xs' (x :: soFar)
def hasDuplicates (xs : List α) :=
  hasDuplicatesAux xs []
def isUniqueAux (xs : List α) (soFar : List α) :=
  match xs with
  | [] ⇒ false
  | x :: xs' ⇒ !List.elem x soFar && isUniqueAux xs' (x :: soFar)
def isUnique (xs : List α) :=
  isUniqueAux xs []
def isAllEvens : List Nat → Bool
  | [] ⇒ true
  | x :: xs ⇒ x % 2 = 0 && isAllEvens xs
```

```
def isRRAux : Tree (Color × α) ⟶ Bool ⟶ Bool := fun t isRedChild ⟹
 match t with
 | .leaf ⟹ true
 | .node l c r ⟹ if c.fst == .red then !isRedChild && isRRAux l true && isRRAux r true else isRRAux l fa
def isRR : Tree (Color × α) ⟶ Bool := fun t ⟹ isRRAux t false
def isBH : Tree (Color × α) ⟶ Nat ⟶ Bool := fun t height ⟹
 match t with
 | .leaf ⟹ height == 0
 | .node l c r ⟹ if c.fst == .red then isBH l height && isBH r height else height >= 0 && isBH l (height
def isRBT : Tree (Color × Nat) ⟶ Nat ⟶ Nat ⟶ Nat ⟶ Bool := fun t height lo hi ⟹
  isRR t = true && isBST t (lo, hi) = true && isBH t height = true
def isRBTNoBST : Tree (Color × Nat) ⟶ Nat ⟶ Bool := fun t height ⟹
  isRR t = true && isBH t height = true
def isAllTwosFold (xs : List Nat) : Bool :=
  List.fold (fun x b ⟹ x == 2 && b) true xs
def isAllTwosEvenLenFold (xs : List Nat) : Bool :=
  List.fold (fun x b ⟹ x == 2 && b) true xs = true
    ∧ List.fold (fun _ b ⟹ !b) true xs
def isEvenLenFold (xs : List α) : Bool :=
  List.fold (fun _ b ⟹ !b) true xs
def isIncreasingByOneFold (xs : List Nat) : Bool :=
  List.fold (fun x b prev ⟹ x == prev + 1 && b x) (fun _ ⟹ true) xs 0
def lengthFold (xs : List α) : Nat :=
  List.fold (fun _ b ⟹ b + 1) 0 xs
def isLengthKAllTwosFold (k : Nat) (xs : List Nat) :=
  List.fold (fun _ b ⟹ b + 1) 0 xs = k
    ∧ List.fold (fun x b ⟹ x == 2 && b) true xs
def isSortedBetweenFold (lo hi : Nat) (xs : List Nat) : Prop :=
  List.fold (fun x b s ⟹ decide (s <= x)
    && decide (x <= hi) && b x) (fun _ ⟹ true) xs lo
def isTrueFold (xs : List α) : Bool :=
  List.fold (fun _ b ⟹ b) true xs
def isAllEvensFold (xs : List Nat) : Bool :=
  List.fold (fun x b ⟹ x % 2 == 0 && b) true xs
def isAllTwos : Tree Nat -> Bool
  | .leaf ⟹ true
  | .node l x r ⟹ x = 2 && isAllTwos l && isAllTwos r
def isBST : Tree Nat -> (Nat × Nat) -> Bool := fun t ⟨lo, hi⟩ ⟹
  match t with
  | .leaf ⟹ true
  | .node l x r ⟹
    (lo <= x && x <= hi) &&
    isBST l ⟨lo, x - 1⟩ &&
    isBST r ⟨x + 1, hi⟩
def isComplete : Tree α -> Nat -> Bool := fun t n ⟹
  match t with
  | .leaf ⟹ n == 0
  | .node l _ r ⟹
```

```
    n > 0 &&
    isComplete l (n - 1) &&
    isComplete r (n - 1)
def isMaxDepth (t : Tree α) (n : Nat) : Bool :=
    match t with
    | .leaf ⇒ true
    | .node l _ r ⇒
      n > 0 &&
      isMaxDepth l (n - 1) &&
      isMaxDepth r (n - 1)
def isIncreasingByOneAux (t : Tree Nat) (prev : Nat) : Bool :=
  match t with
  | .leaf ⇒ true
  | .node l x r ⇒
    x == prev + 1 &&
    isIncreasingByOneAux l x &&
    isIncreasingByOneAux r x
def isIncreasingByOne (t : Tree Nat) : Bool :=
  isIncreasingByOneAux t 0
def isNonempty : Tree α -> Bool
  | .leaf ⇒ false
  | .node l _ r ⇒ true && isNonempty l && isNonempty r
def isAllTwosFold (t : Tree Nat) : Bool :=
  Tree.fold (fun bl x br ⇒ x == 2 && bl && br) true t
def isBSTFold (lo hi : Nat) (t : Tree Nat) : Bool :=
  Tree.fold
      (fun bl x br s ⇒
        match s with
        | (sl, sr) ⇒
          (decide (sl <= x) &&
            decide (x <= sr)) &&
            bl (sl, x - 1) && br (x + 1, sr))
      (fun _ ⇒ true) t (lo, hi)
def isCompleteFold (n : Nat) (t : Tree Nat) : Bool :=
  Tree.fold (fun bl _ br s ⇒ decide (s > 0) &&
  bl (s - 1) &&
  br (s - 1)) (fun s ⇒ s == 0) t n
def isMaxDepthFold (t : Tree Nat) (n : Nat) : Bool :=
  Tree.fold (fun bl _ br s ⇒ decide (s > 0) &&
  bl (s - 1) &&
  br (s - 1)) (fun _ ⇒ true) t n
def isIncreasingByOneFold (t : Tree Nat) : Bool :=
  Tree.fold
    (fun bl x br prev ⇒ x == prev + 1 && bl x && br x)
    (fun _ ⇒ true) t 0
def isNonemptyFold (t : Tree α) : Bool :=
  Tree.fold (fun _ _ _ ⇒ true) false t
def isBalanced : Tree Nat -> Nat -> Bool := fun t height ⇒
```

```
  match t with
  | .leaf ⇒ height <= 1
  | .node l _ r ⇒
    height > 0 &&
    isBalanced l (height - 1) &&
    isBalanced r (height - 1)
def isAVL (height lo hi : Nat) (t : Tree Nat) : Bool :=
  isBST t (lo, hi) && isBalanced t height
def isAVLFold (height lo hi : Nat) (t : Tree Nat) : Bool :=
  Tree.fold
      (fun bl x br bounds ⇒
        match bounds with
        | (sl, sr) ⇒ decide (sl <= x) && decide (x <= sr)
          && bl (sl, x - 1) && br (x + 1, sr))
      (fun _ ⇒ true) t (lo, hi) = true
    ∧
    Tree.fold
      (fun bl _ br h ⇒ decide (h > 0) && bl (h - 1) && br (h - 1))
      (fun h ⇒ decide (h <= 1)) t height
def isRRFold (t : Tree (Color × α)) : Bool :=
  Tree.fold
    (fun bl c br isRedChild ⇒ if c.fst == .red then !isRedChild && bl true && br true else bl false && b
    (fun _ ⇒ true)
    t
    false
def isBHFold (t : Tree (Color × α)) (height : Nat) : Bool :=
  Tree.fold
    (fun bl c br h ⇒ if c.fst == .red then bl h && br h else h >= 0 && bl (h - 1) && br (h - 1))
    (fun h ⇒ h == 0)
    t
    height
def isRBTFold (height lo hi : Nat) (t : Tree (Color × Nat)) : Bool :=
  isBHFold t height = true && isRRFold t = true && isBSTFold t (lo, hi) = true
def isRBTNoBSTFold (height : Nat) (t : Tree (Color × Nat)) : Bool :=
  isBHFold t height = true && isRRFold t = true
def isGoodNat (n : Nat) : Bool :=
  n == 0 || n == 1
def isGoodAtom : Atom -> Bool
  | .atm n _ ⇒ isGoodNat n
def isGoodStackFold (s : Stack) (n : Nat) : Bool :=
  Stack.fold (fun s ⇒ s == 0)
    (fun x acc s ⇒ isGoodAtom x && acc (s - 1))
    (fun pc acc s ⇒ isGoodAtom pc && acc (s - 1)) s n
def isGoodStack (s : Stack) (n : Nat) : Bool :=
  match s with
  | .mty ⇒ n == 0
  | .cons x s' ⇒ (n > 0 && isGoodAtom x) && isGoodStack s' (n - 1)
  | .ret_cons pc s' ⇒ (n > 0 && isGoodAtom pc) && isGoodStack s' (n - 1)
```

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

E Extended Tables of Benchmarks

E.1 Main

The following table lists the 32 benchmarks in the Main benchmark set.

The value being generated is **v**; other variables are universally quantified unless specified. External definitions (e.g., `isBST`) are presented in Appendix D. We list several measures of the complexity of the benchmark: First, each type being generated lists the number of constructors for the ADT, which indicates case splitting possibilities in the search. Next, for each benchmark we also count the number of functions used in the specification, e.g., `isAllTwosEvenLength` is specified using two auxiliary functions, `isAllTwos` and `isEvenLength`. Finally, we tally the number of recursive calls across all these functions.

Type	ADT Cases	Predicate	no. functions in spec	no. recursive calls (total)
Nat	2	`v = 2`	0	0
		`2 = v`	0	0
		`v = 2 ∨ v = 5`	0	0
		`v = 2 ∨ v = 5 ∧ True`	0	0
		`∃ a, a = 3 ∧ v = a + 1`	0	0
		`5 <= v ∧ v <= 10`	0	0
		`v > 5`	0	0
		`lo <=v ∧ v <= hi`	0	0
		`v = 0∨ lo <= v ∧ v <= hi`	0	0
List Nat	2	`isAllTwos v = true`	1	1
		`isAllTwosEvenLen v = true`	3	2
		`isEvenLen v = true`	1	1
		`isIncreasingByOne v = true`	2	1
		`List.length v = k`	0	0
		`isLengthKAllTwos k v = true`	2	1
		`isSortedBetween v (lo, hi) = true`	1	1
		`isAllEvens v = true`	1	1
		`isTrue v = true`	1	1
		`isUnique v`	2	1
		`hasDuplicates v`	2	1
Tree Nat	2	`isAllTwos v = true`	1	2
		`isBST v (lo, hi) = true`	1	2
		`isComplete v n = true`	1	2
		`isMaxDepth v n = true`	1	2
		`isIncreasingByOne v = true`	2	2
		`isNonempty v = true`	1	2
		`isAVL height lo hi v = true`	3	4
		`isRBTNoBST height v = true`	4	8
		`isRBT height lo hi v = true`	5	10
Stack	3	`isGoodStack v n = true`	3	2
Term	4	`isWellScoped v 0 = true`	1	3
		`isWellTyped Γ v`	2	3

The following table lists the runtimes of Palamedes on the Main benchmarks. Means are presented with standard deviations in parentheses. Generators above the line are assume-free. The four generators below the line are not: AVL and RBT generators are known to need backtracking (e.g., the generator in Prinz and Lampropoulos [45]). The generator of a range is not assume-free because the assume introduced by the S-ChoosePartial rule described in Section 3.3 is not optimized out because it is not surrounded by a pick.

Predicate	Type	Time (s)
`v = 2`	`Nat`	0.04 (0.00)
`2 = v`	`Nat`	0.05 (0.00)
`v = 2 ∨ v = 5`	`Nat`	0.08 (0.01)
`v = 2 ∨ v = 5 ∧ True`	`Nat`	0.07 (0.00)
`∃ a, a = 3 ∧ v = a + 1`	`Nat`	0.05 (0.00)
`5 <= v ∧ v <= 10`	`Nat`	0.07 (0.00)
`v > 5`	`Nat`	0.06 (0.00)
`v = 0 ∨ lo <= v ∧ v <= hi`	`Nat`	0.12 (0.01)
`isAllTwos v = true`	`List Nat`	0.82 (0.02)
`isAllTwosEvenLen v = true`	`List Nat`	2.36 (0.4)
`isEvenLen v = true`	`List Nat`	2.15 (0.04)
`isIncreasingByOne v = true`	`List Nat`	1.46 (0.03)
`List.length v = k`	`List Nat`	1.86 (0.03)
`isLengthKAllTwos k v = true`	`List Nat`	3.38 (0.05)
`isSortedBetween v (lo, hi) = true`	`List Nat`	1.61 (0.03)
`isAllEvens v = true`	`List Nat`	0.92 (0.02)
`isTrue v = true`	`List Nat`	2.14 (0.04)
`isUnique v`	`List Nat`	FAIL in 1.9s
`hasDuplicates v`	`List Nat`	FAIL in 1.75s
`isAllTwos v = true`	`Tree Nat`	2.17 (0.04)
`isBST v (lo, hi) = true`	`Tree Nat`	1.53 (0.04)
`isComplete v n = true`	`Tree Nat`	1.57 (0.03)
`isMaxDepth v n = true`	`Tree Nat`	1.65 (0.04)
`isIncreasingByOne v = true`	`Tree Nat`	1.35 (0.03)
`isNonempty v = true`	`Tree Nat`	1.40 (0.03)
`isGoodStack v n = true`	`Stack`	2.76 (0.06)
`isWellScoped v 0 = true`	`Term`	1.64 (0.03)
`isWellTyped Γ v`	`Term`	2.47 (0.05)
`lo <= v ∧ v <= hi`	`Nat`	0.06 (0.00)
`isAVL height lo hi v = true`	`Tree Nat`	8.29 (0.18)
`isRBTNoBST height v = true`	`Tree Nat`	8.80 (0.22)
`isRBT height lo hi v = true`	`Tree Nat`	50.4 (0.89)

E.2 Cobb

Predicate	Type	Maximal Sketch Time (s)	Minimal Sketch Time (s)
`isSortedBetween v (lo, hi) = true`	`List Nat`	0.58	5.58
`isAllEvens v = true`	`List Nat`	2.56	10.35
`isTrue v = true`	`List Nat`	Insufficient Lemmas	
`isIncreasingByOne v = true`	`List Nat`	0.60	5.33
`isAllTwos v = true`	`List Nat`	1.11	3.32
`isAllTwosEvenLen v = true`	`List Nat`	0.69	Timeout
`isEvenLen v = true`	`List Nat`	0.72	Timeout
`List.length v = k`	`List Nat`	0.61	2.16
`isLengthKAllTwos k v = true`	`List Nat`	0.67	2.70
`isUnique v`	`List Nat`	0.52	1.51
`hasDuplicates v`	`List Nat`	0.66	3.92
`isMaxDepth v n = true`	`Tree Nat`	1.17	8.29
`isComplete v n = true`	`Tree Nat`	0.7	4.20
`isIncreasingByOne v = true`	`Tree Nat`	1.54	191.74
`isNonempty v = true`	`Tree Nat`	3.48	11.82
`isAllTwos v = true`	`Tree Nat`	2.99	16.48
`isRBTNoBST height v = true`	`Tree Nat`	39.62	64.9
`isRBT height lo hi v = true`	`Tree Nat`	Insufficient Lemmas	
`isBST v (lo, hi) = true`	`Tree Nat`	46.08	736.76
`isAVL height lo hi v = true`	`Tree Nat`	Insufficient Lemmas	
`isWellTyped Γ v`	`Term`	5.6	577.86
`isWellScoped v 0 = true`	`Term`	Insufficient Lemmas	
`isGoodStack v n = true`	`Stack`	Insufficient Lemmas	
`v = 2`	`Nat`	Immediate failure (no sketch)	
`2 = v`	`Nat`	Immediate failure (no sketch)	
`v = 2 ∨ v = 5`	`Nat`	Immediate failure (no sketch)	
`v = 2 ∨ v = 5 ∧ True`	`Nat`	Immediate failure (no sketch)	
`∃ a, a = 3 ∧ v = a + 1`	`Nat`	Immediate failure (no sketch)	
`5 <= v ∧ v <= 10`	`Nat`	Immediate failure (no sketch)	
`v > 5`	`Nat`	Immediate failure (no sketch)	
`v = 0∨ lo <= v ∧ v <= hi`	`Nat`	Immediate failure (no sketch)	
`lo <= v ∧ v <= hi`	`Nat`	Immediate failure (no sketch)	

E.3 QuickChick

Classification of different benchmarks on the translation process of the MAIN benchmarks. Natural denotes that the natural direct translation worked, Merging denotes that they needed to manually invoke merging to obtain an efficient generator, Tweaking denotes that the order of recursive premises matters for performance so tweaking might be required, Out-of-Scope is an existential which is out of QuickChick's automation scope.

Predicate	Classification
`v = 2`	Natural
`2 =v`	Natural
`v = 2 ∨ v = 5`	Natural
`v = 2 ∨ v = 5 ∧ True`	Merging
`∃ a, a = 3 ∧ v = a + 1`	Out-of-Scope
`5 <= v ∧ v <= 10`	Merging
`v > 5`	Natural
`v = 0 ∨ lo <= v ∧ v <= hi`	Merging
`isAllTwos v = true`	Natural
`isAllTwosEvenLen v = true`	Merging
`isEvenLen v = true`	Natural
`isIncreasingByOne v = true`	Natural
`List.length v = k`	Natural
`isLengthKAllTwos k v = true`	Merging
`isSortedBetween v (lo, hi) = true`	Merging
`isAllEvens v = true`	Natural
`isTrue v = true`	Natural
`isUnique v`	Natural
`hasDuplicates v`	Natural
`isAllTwos v = true`	Natural
`isBST v (lo, hi) = true`	Merging
`isComplete v n = true`	Natural
`isMaxDepth v n = true`	Natural
`isIncreasingByOne v = true`	Natural
`isNonempty v = true`	Natural
`isGoodStack v n = true`	Natural
`isWellScoped v 0 = true`	Natural
`isWellTyped Γ v`	Tweaking
`lo <= v ∧ v <= hi`	Merging
`isAVL height lo hi v = true`	Merging
`isRBTNoBST height v = true`	Natural
`isRBT height lo hi v = true`	Merging

F 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 () ⇒
            -- $\tau$ = .unit
            pure TermF.unitStep)
          (fun τ1 τ2 () ⇒
            -- $\tau$ = .arrow $\tau$1 $\tau$2
            pure (TermF.absStep τ1 (τ2, τ1 :: 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)
```

G Synthesized Generator of Well-Typed Terms

```
def genWellTyped (Γ : List Ty) : Gen Term := by
  let τ <- arbTy; Term.unfold
    (fun (τ, Γ) ⇒ do
      let step <- caseTy τ
        (fun () ⇒                                          -- $\tau$ = .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 ()  ⇒                                    -- $\tau$ = .arrow $\tau$1 $\tau$2
          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)
```

H Synthesized Generator of AVL Trees

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

I Manually Written Generators of AVL Trees

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

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

J More Side-by-Side Generator Comparisons

J.1 One Or In Range

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

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

J.2 Complete Tree

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

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

J.3 Sorted Between

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

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

J.4 Length K, All Twos

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

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

Received 2025-11-13; accepted 2026-04-03