Tuning Random Generators

Property-Based Testing as Probabilistic Programming

RYAN TJOA, University of Washington, USA
POORVA GARG, University of California, Los Angeles, USA
HARRISON GOLDSTEIN, University of Maryland, USA
TODD MILLSTEIN, University of California, Los Angeles, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
GUY VAN DEN BROECK, University of California, Los Angeles, USA

Property-based testing validates software against an executable specification by evaluating it on randomly generated inputs. The standard way that PBT users generate test inputs is via *generators* that describe how to sample test inputs through random choices. To achieve a good distribution over test inputs, users must *tune* their generators, i.e., decide on the weights of these individual random choices. Unfortunately, it is very difficult to understand how to choose individual generator weights in order to achieve a desired distribution, so today this process is tedious and limits the distributions that can be practically achieved.

In this paper, we develop techniques for the automatic and offline tuning of generators. Given a generator with undetermined *symbolic weights* and an *objective function*, our approach automatically learns values for these weights that optimize for the objective. We describe useful objective functions that allow users to (1) target desired distributions and (2) improve the diversity and validity of their test cases. We have implemented our approach in a novel discrete probabilistic programming system, Loaded Dice, that supports differentiation and parameter learning, and use it as a language for generators. We empirically demonstrate that our approach is effective at optimizing generator distributions according to the specified objective functions. We also perform a thorough evaluation on PBT benchmarks, demonstrating that, when automatically tuned for diversity and validity, the generators exhibit a 3.1–7.4× speedup in bug finding.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*; • **Mathematics of computing** → *Probabilistic inference problems*; *Automatic differentiation.*

Additional Key Words and Phrases: Random Generation, Property-Based Testing, Probabilistic Programming, Automatic Differentiation

ACM Reference Format:
Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd Millstein, Benjamin C. Pierce, and Guy Van den Broeck. 2025. Tuning Random Generators: Property-Based Testing as Probabilistic Programming. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 304 (October 2025), 28 pages. https://doi.org/10.1145/3763082

1 Introduction

Property-based testing (PBT) is a powerful [2, 3, 29] and widely-studied [22, 27, 28] software testing technique that validates a system under test with respect to an executable specification by

Authors' Contact Information: Ryan Tjoa, University of Washington, Seattle, USA, rtjoa@cs.washington.edu; Poorva Garg, University of California, Los Angeles, USA, poorvagarg@cs.ucla.edu; Harrison Goldstein, University of Maryland, College Park, USA, me@harrisongoldste.in; Todd Millstein, University of California, Los Angeles, USA, todd@cs.ucla.edu; Benjamin C. Pierce, University of Pennsylvania, Philadelphia, USA, bcpierce@seas.upenn.edu; Guy Van den Broeck, University of California, Los Angeles, USA, guyvdb@cs.ucla.edu.

ACM 2475-1421/2025/10-ART304
https://doi.org/10.1145/3763082

evaluating it on many randomly generated inputs. For example, when testing a sorting function, a user might write the property

```
∀l. isSorted (sort l),
```

which specifies that the result of sorting a list should be sorted. To check this property, the PBT framework generates hundreds or thousands of inputs to the property (lists `l`) and checks the statement with respect to each one. Since testing performance is entirely dependent on the distribution of test inputs, a great deal of PBT research has focused on how to quickly generate inputs that find more bugs, faster [17, 22].

The standard way that PBT users generate test inputs is via *generators* — programs that describe how to sample test inputs from some random distribution. There has been extensive research both on domain-specific languages for manually constructing generators [31] and on methods for automatically deriving generators from data type definitions [36] or inductive relations [32].

But despite all this flexibility and automation, a key challenge remains: *generator tuning*. In order to achieve a good distribution of test inputs—one where "interesting" inputs appear often—the programmer has to manually decide on the weights of the individual random choices that are made as the generator executes.

For example, the following generator is the one a developer might try writing down when testing the above `sort` property. We write it in our Loaded Dice[1] probabilistic programming language (Section 4), which is an embedded domain-specific language in Julia; `@match` is a Julia macro defined by Loaded Dice to implement pattern matching.

```
genList(sz) = # generates lists up to length sz
  @match sz (
    0 → Nil(),
    S(sz') → oneOf [ Nil(), Cons(genNat(), genList(sz')) ])
```

If a developer inspects the test cases produced by this generator, they will quickly notice an obvious problem — 50% of the generated test cases will be empty lists! This is because the `oneOf` combinator in the generator makes a uniform random choice between the two constructors for the data type. Half of the time, it chooses `Nil`, and the other half of the time, it chooses `Cons`. Note that this means not only are half the lists empty, but half of the rest are length 1, and so on.

Even if it is obvious to the developer that this distribution is a poor choice for testing, it may not be obvious how to improve the situation. One could use `freq`, a replacement for `oneOf` that allows the user to manually add weights to the random choice. For example, the following use of `freq` chooses `Cons` two-thirds of the time.

```
freq [1 ⇒ Nil(), 2 ⇒ Cons(genNat(), genList(sz'))]
```

But which weights should the programmer use? In general, and especially as generators get more complicated, it can be a significant challenge to understand how changing weights changes the final distribution. With the above, the distribution of list lengths is roughly[2] the geometric distribution with success probability two-thirds. Reasoning about the distributions of more complex generators, and how they depend on the weights, quickly increases in difficulty. Indeed, recent work on PBT usability [23] cited tuning as a source of "mental strain" for developers who felt like they needed to "study probability and statistics" to understand how to tune a generator to suit their needs.

In this paper, we address this issue by providing developers with techniques for *automatically* tuning generators. Concretely, users can write down generators with *symbolic weights* that are not yet determined, then specify an *objective function* that the weights should attempt to optimize. We

[1] Loaded Dice is available at https://github.com/Tractables/Alea.jl/tree/loaded-dice.

[2] It differs from the geometric distribution because it is truncated at the initial `sz`, as `Nil` is always chosen when `sz` is 0.

present an offline approach to automatically learn values for these weights to optimize for a given objective function.

Our approach is flexible enough to handle a wide variety of objective functions. If the developer has an intuition about the distribution they want (e.g., that the distribution of lengths of generated lists should be uniform) they can simply optimize the generator to try to match that distribution. If they don't know the precise distribution they're after, they can instead favor diversity of test cases by optimizing for *entropy* [46], a standard metric for diversity. Finally, if the developer has a notion of "validity" that they want to maintain, they can combine entropy with adherence to some specification of validity.

Our approach automatically tunes PBT generators to optimize objective functions by expressing the generators as programs in an extension of Dice [26], a discrete probabilistic programming language (PPL). PPLs and generator languages both deal with randomness, but, for our purposes, Dice has a significant advantage: it can perform exact probabilistic inference, computing a representation of the full distribution of a given generator. Furthermore, the inference strategy in Dice is differentiable, so we can use gradient descent to optimize symbolic generator weights for a given objective. We design and implement Loaded Dice, an extension of Dice that supports differentiation and parameter learning, and use it as a language for generators.

To make tuning feasible, we address two key performance challenges. First, probabilistic inference for discrete PPLs is #P-hard in general [16], which in turn makes computing gradients #P-hard as well. We address this problem by choosing our PPL carefully: Dice compiles to binary decision diagrams (BDDs) that naturally exploit program structure to scale probabilistic inference. Since the computation of gradients in Loaded Dice happens on the same BDDs, it can leverage the same scaling benefits. The second performance challenge arises from the fact that the naïve way to compute our proposed objective functions requires enumerating the whole space of possible test cases. This is infeasible, so we adapt a scalable gradient estimation technique, REINFORCE [51], to our context of generator tuning. At a high level, REINFORCE allows us to avoid this large enumeration by replacing it with sampling.

With these elements in place, we present multiple examples that demonstrate the effectiveness of our approach in steering the distributions of the generators. We also perform a thorough evaluation on PBT benchmarks, demonstrating that, when tuned for diversity and validity, the generators lead to a 3.1–7.4× speedup in bug finding.

Following a high-level overview in Section 2, we offer the following contributions:

- We describe a space of generator-independent objective functions that can target a specific distribution or increase the diversity and validity of the generator (Section 3).
- We describe the design and implementation of Loaded Dice, a PPL that allows weights to be learned for these objectives by extending Dice with automatic differentiation (Section 4).
- We show techniques for constructing generators more amenable to tuning, and how to derive such generators from inductive type definitions (Section 5).
- We present training techniques to achieve these objectives in practice. In particular, we adapt REINFORCE, a gradient estimation technique, to the context of generator tuning, in order to efficiently optimize for entropy-based objectives (Section 6).
- To evaluate our approach, we use Loaded Dice to tune a diverse collection of type-based generators for validity and diversity and to tune a handwritten STLC generator for a particular distribution, improving the speed at which they find bugs on existing benchmarks (Section 7).

2 Overview

In this section, we overview our approach, focusing in particular on how it can benefit PBT users.

2.1 The Basics of Generator Tuning

Since the beginning of PBT, in QuickCheck [18], generators have been a core part of the PBT process. PBT frameworks provide a domain-specific language (DSL) for expressing and combining generators, and programmers can use that language to design arbitrarily complicated distributions of test inputs for their programs.

While the power of generator DSLs is generally a significant benefit, the complexity of PBT generators leads to some important challenges. The key challenge we focus on in this paper is *tuning*, which is the process of choosing the weights with which different random choices in the generator are made.

To illustrate tuning and demonstrate why it is difficult, we start with a toy example. Consider this Loaded Dice generator of characters from `'a'`–`'e'`:

```
G = freq([
  θ1 ⇒ freq([θ2 ⇒ 'a', θ3 ⇒ 'b', θ4 ⇒ 'c']),
  θ5 ⇒ freq([θ6 ⇒ 'c', θ7 ⇒ 'd', θ8 ⇒ 'e'])])
```

The generator `G` uses the `freq` combinator to make weighted random choices between different options; each θ is a placeholder for a number that decides the relative weight of that particular choice. For example, if θ_2, θ_3, and θ_4 were all 1, the `freq` containing them would make a uniform choice. If θ_2 were changed to 2, the value `'a'` would be chosen twice as often as `'b'` or `'c'`.

Now, suppose we want to ensure that `G` has a uniform distribution over the five characters — that is, that each is sampled 20% of the time. We encourage readers to take a moment to try to come up with values for θ_1–θ_8 that produce the appropriate distribution. Even for this very simple example, it is not totally obvious!

Our approach entirely automates this reasoning. The user can simply write a target distribution and then ask that the generator be trained to match that distribution:

```
target = ['a' ⇒ 0.2, 'b' ⇒ 0.2, 'c' ⇒ 0.2, 'd' ⇒ 0.2, 'e' ⇒ 0.2]
objective = -kl_divergence(target, G)
```

We discuss KL divergence later in detail; for now, read line 2 as "pick weights in `G` such that the final distribution is as close as possible to `target`." Given these inputs, our approach automatically learns weights proportional to $\{\theta_1 \mapsto 1,\ \theta_2 \mapsto 2,\ \theta_3 \mapsto 2,\ \theta_4 \mapsto 1,\ \theta_5 \mapsto 1,\ \theta_6 \mapsto 1,\ \theta_7 \mapsto 2,\ \theta_8 \mapsto 2\}$, which achieves the desired distribution.

```
@type Color = R() | B()
@type Tree = Leaf() | Branch(Color, Tree, Nat, Nat, Tree)

genColor() = freq([θred ⇒ R(), 1 - θred ⇒ B()])

genTree(size) =
  @match size (
    0 → Leaf(),
    S(n) → (
      w = @match size ( 1 ⇒ θ1, 2 ⇒ θ2, 3 ⇒ θ3, 4 ⇒ θ4, 5 ⇒ θ5 );
      freq([w ⇒ Leaf(),
            1-w ⇒ Branch(genColor(), genTree(n), genNat(), genNat(), genTree(n))])))

G = genTree(5)
```

Fig. 1. A generator of (not necessarily valid) red-black trees, using symbolic weights. The macros `@type` and `@match`, provided by the Loaded Dice embedding in Julia, implement algebraic datatypes and pattern matching. The match expression computing `w` allows the weights to depend on `size`, as described in Section 5.1.1.

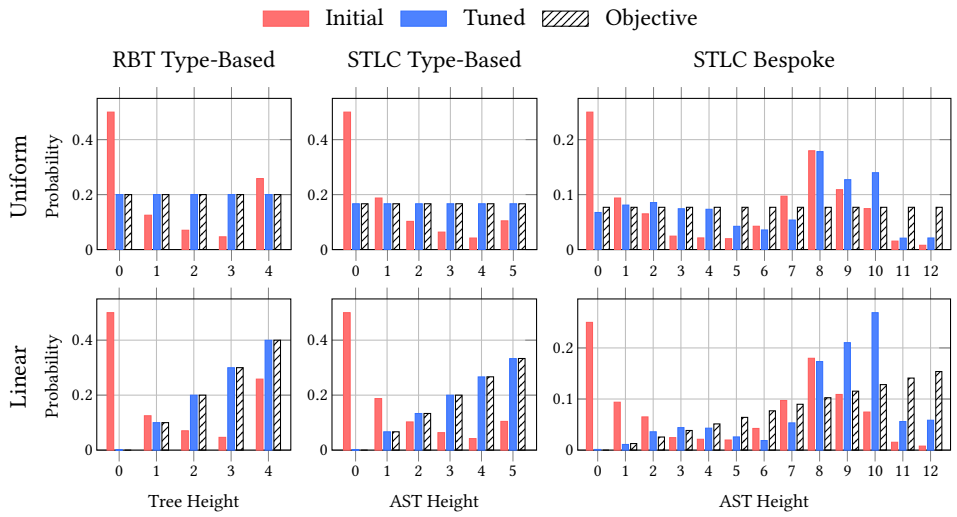


Fig. 2. Tuning the distributions of heights of generated values to be uniform or linear for several generators. The top row trains the generators to have a uniform distribution over heights, and the bottom row trains the generators to have a linear distribution over heights.[4]

2.2 Approximating Known Distributions

Next, we move on to a more realistic example, continuing to explore how our approach allows developers to tune generators according to a concrete desired distribution.

Consider the generator in Figure 1, which generates random color-labeled binary trees (some subset of these trees will be valid red-black trees [24], but this generator does not ensure that invariant). The function `genTree` takes a maximum tree `size` and then generates trees up to that size. If `size` is non-zero, `genTree` makes a random choice: with probability `w` it generates a leaf, and with probability `1 - w` it generates a color, key, and value for an internal node, then recurses to generate the two child trees with reduced `size`. If `size` is zero, then it always produces a leaf.

Starting from this generator, a developer might have some ideas for what they would like the distribution of trees to look like. For example, they may tune the generator to produce relatively few very small trees (height 1 and 2) and proportionally more large trees (height 4 and 5). They can specify their desired distribution over heights by using a `height` function and `kl_divergence`:

```
target = [1 ⇒ 0.1, 2 ⇒ 0.1, 3 ⇒ 0.2, 4 ⇒ 0.3, 5 ⇒ 0.3]
objective = -kl_divergence(target, height(G))
```

This is similar to the example above, but now both the generator and the objective are significantly more realistic: the generator is modeled after one that appears often in the PBT literature, and the objective adheres to a common PBT principle that larger test inputs often find more bugs.[3]

Figure 2 demonstrates this process on a wider range of examples. We show the results of tuning three generators: the above generator for color-labeled binary trees, a generator for terms in the simply-typed lambda calculus (STLC) that may or may not be well-typed, and a "bespoke" handwritten generator for well-typed STLC terms (generator shown in Appendix B). For each,

[3] As observed by Shi et al. [47] and shown in Section 7, it can also be beneficial to tune for *smaller* inputs.

[4] The tuned distributions of the AST heights in the STLC bespoke generator do not exactly match the target distribution, but they are closer to their objectives: tuning improves KL divergence from 0.44 to 0.22 for the uniform target distribution, and from 0.92 to 0.27 for the linear target distribution.

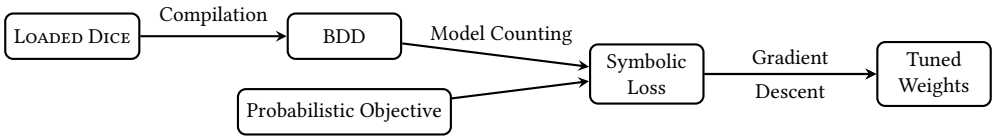


Fig. 3. Tuning Generators in Loaded Dice.

we tune the distributions to have either a uniform or a linear relationship between data structure height and sampling frequency. The charts show that the tuned generators (in blue) match the target distribution (in gray) much better than the untuned[5] versions (in red).

2.3 Under the Hood

Sections 3, 4, and 6 discuss the technical details of our approach and its implementation in significant detail; here we simply give a high level picture.

Our key observation is that by implementing generators in a probabilistic programming language, we get easy access to algorithms that can be used to automate tuning. In particular, we choose Dice [26] as our starting point because it provides scalable procedures for *exact probabilistic inference* — computing a closed-form representation of the whole generator distribution. This means that if we implement a generator in Dice, we can compute precisely how well that generator matches a distribution requested by the user.

We extend Dice to a probabilistic programming system called Loaded Dice with two significant additions. First, in Dice all weights must be concrete numbers; Loaded Dice instead has *symbolic weights* (the θs in the generators above), which allow the programmer to choose points in the generator where weights should be learned automatically. Second, Loaded Dice extends Dice's inference algorithm to compute *gradients*; we can compute not only the distribution of a generator, but also how the symbolic weights should be changed to maximize a given objective function.

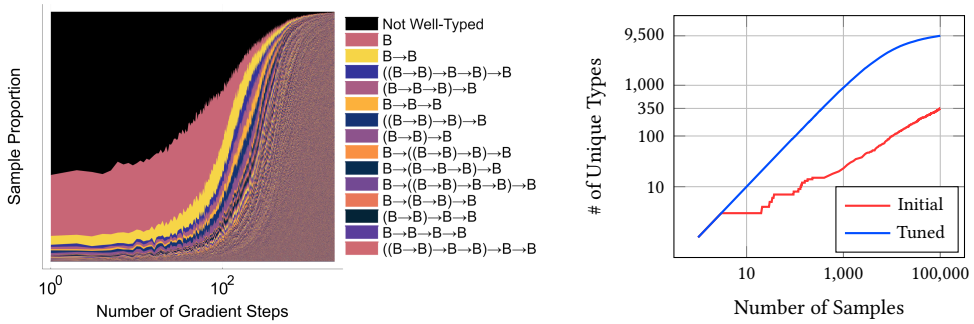
With these features in place, Figure 3 shows the workflow of our approach. A generator in Loaded Dice with symbolic weights is first compiled to a binary decision diagram (BDD) that represents all possible executions of the program. The benefit of this representation is that we can then perform exact probabilistic inference on the program as a linear pass over the BDD [26]. Specifically, the procedure of *weighted model counting* on the BDD generates expressions over the symbolic weights that correspond to the probability distribution of the generator. These expressions are then plugged into the user's desired objective function, and the result is a differentiable expression over the symbolic weights in the generator that can be optimized using gradient descent. The system repeatedly computes the gradient of the generator with respect to the objective function and then nudges the generator weights in the appropriate direction. In the end, the training process is likely to stabilize on a choice of weights that gets close to the desired distribution [11].

2.4 More General Tuning Objectives

Matching developer-specified distributions is certainly useful, but in real-world settings a developer might not actually know what distribution will be best for testing. To address this situation, we have identified two generally useful properties of a distribution of test cases that we can use as generic tuning objectives.

One natural objective that a developer may want for their generator is maximizing the diversity or *entropy* of the generated values. For generators producing unconstrained values, this can be an ideal

[5]Throughout this paper, an "untuned" generator is one in which each random choice is uniformly distributed.



(a) A visualization of how the distribution over types changes as we tune the weights. The x-axis uses log scaling.

(b) Cumulative unique types throughout sampling, before and after tuning. Both axes use log scaling.

Fig. 4. Results for tuning an STLC generator for unique types using specification entropy. For brevity, the legend of (a) shows only the most common types and abbreviates "Bool" as "B."

shortcut to a balanced distribution. For example, maximizing the entropy of the `'a'–'e'` generator from Section 2.1 gives the same weights as the more explicitly defined uniform distribution.

But often, values are constrained by a *validity predicate* — for example, the color-labeled tree example from Section 2.2 may be expected to produce valid red-black trees for the purposes of testing functions that require red-black tree validity as an invariant. For these situations, maximizing entropy alone is not sufficient. We therefore define a notion of *specification entropy*, which attempts to simultaneously optimize entropy and validity. Figure 4 shows the results of tuning an STLC generator to increase the entropy of the types of generated terms, and simultaneously increase the likelihood of well-typedness. Figure 4a visualizes how the distribution over types changes over time as we tune the weights; Figure 4b contrasts the initial generator with the tuned version by comparing the number of unique types each generates as the number of samples increases. The trained version produces terms with far more diverse and interesting types. At the beginning of tuning, ill-typed terms and terms of type bool comprise 66% and 25% of samples, respectively. After tuning, neither any single type nor ill-typed terms comprise more than 0.5% of samples.

3 Objective Functions

Users of property-based testing sometimes have intuitions about what specific generator distributions are desirable for their use case [21]. This section describes how these intuitions can be expressed as objective functions for automated generator tuning in our approach. We first provide some preliminaries, and then Section 3.2 describes how a user can specify an objective function to tune for a target distribution. Then, Section 3.3 describes an objective function to improve the diversity and validity of their test cases.

3.1 Preliminaries and Notation

The random choices made by a generator induce a particular distribution over the test cases it can produce. Let G denote a generator for test cases of type T with $n \in \mathbb{N}$ symbolic weights. We represent an *assignment* to its weights as $w \in [0,1]^n$. Then, we denote the probability distribution induced by G instantiated with those weights as $p_{G,w}$.

Now, for automated generator tuning, we need a measure of how good the generator distribution is. For this purpose, we define an *objective function* as follows:

Definition 1 (objective function). *Given a generator G with n symbolic weights, an objective function $f : [0,1]^n \to \mathbb{R}$ is defined such that for two assignments of weights, w and w', if $f(w) > f(w')$ then the user prefers distribution $p_{G,w}$ over $p_{G,w'}$.*

Example 1. *If G generates characters* `'a'-'e'`*, the objective function $f(w) = p_{G,w}(\texttt{'b'})$ simply maximizes the probability of generating* `'b'`*.*

We provide two useful families of objective functions, which we specify below. We introduce the target objective function to tune for a particular distribution and the specification entropy objective function to improve the diversity and validity of generated test cases.

3.2 Objective Function to Target a Distribution

As stated in Section 2, PBT users sometimes desire a particular distribution over a feature of their generated test cases (say, one might want a generator for RBTs to produce a uniform distribution over tree heights). In fact, there are numerous tools that record the generator distribution to make it easier for the user to visualize [23, 33, 49], but the user still has to manually update and reason about the weights to adjust their distribution. Through automated generator tuning, our approach optimizes weights such that the generator distribution approaches the user's desired distribution.

To define an objective function for this task, we first capture how a generator can induce a distribution over a feature of its generated test cases through the following definition:

Definition 2 (push-forward of generator distribution). *The push-forward of a generator distribution $p_{G,w}$ over type T through a function $g : T \to T'$ is the probability distribution $p_{G,w,g}$ over type T' such that*

$$\forall t' \in T', \quad p_{G,w,g}(t') = \sum_{t \in T, g(t) = t'} p_{G,w}(t).$$

Example 2. *Let G be a generator over lists such that $p_{G,w}([1,2]) = 0.7$ and $p_{G,w}([2,3]) = 0.3$ and g be the length function for lists. Then, by the above definition, $p_{G,w,g}(2) = 1$.*

Now, the objective function that aims for a particular distribution should minimize the distance between the generator distribution and the target distribution. To capture this notion, we use KL divergence [30] as the measure of how much one probability distribution differs from the other and define the *target objective function* as follows:

Definition 3 (target objective function). *Given a generator G for test cases of type T, a function $g : T \to T'$, and a target distribution $\tilde{p}$ over values of type T', the target objective function is defined as the negative KL divergence between the target distribution and the push forward of the generator distribution through g.*

$$Target(w) := -KLD(\tilde{p}, p_{G,w,g}) = -\sum_{t' \in T'} \tilde{p}(t') \log \frac{\tilde{p}(t')}{p_{G,w,g}(t')}$$

Example 3. *Let G be a generator for red-black trees with symbolic weights w. Let g be a function that takes as input a red-black tree and outputs its height. Let the user-specified target distribution $\tilde{p}$ be defined over the tree height as $\{2 \to 0.3, 4 \to 0.7\}$, then the target objective function would be*

$$Target(w) = -0.3 \log \frac{0.3}{p_{G,w,g}(2)} - 0.7 \log \frac{0.7}{p_{G,w,g}(4)}.$$

We demonstrate the effectiveness of the target objective function in Figure 2 where we tune three different generators for uniform and linear distributions over a feature of the test cases. In

Section 7, we use the target objective function to leverage insight from Shi et al. [47] that smaller STLC terms find bugs faster, resulting in improved bug-finding speed.

To summarize, in this section we have shown how to turn a user-specified distribution into an objective function that we can optimize for. This is useful when the user has intuition about what the generator distribution should look like.

3.3 Objective Function to Improve Diversity and Validity

If a user does not have a target distribution in mind, they may instead want to tune the generator weights to improve the diversity of their test cases and increase the number of valid ones. This has the potential to speed up testing by exercising a wider variety of program configurations and reducing the time spent generating invalid inputs. This section describes the objective functions to optimize for these distributional properties and how one can combine them.

3.3.1 Targeting Diverse Generations. In this section we examine how to improve the diversity of the test cases produced by a generator. For this purpose, we define the entropy objective function using the information-theoretic notion of entropy [46] of a probability distribution.

Definition 4 (entropy objective function). *The entropy objective function for a generator G is defined as the entropy of its generator distribution $p_{G,w}$.*

$$\mathit{Entropy}(w) := H(p_{G,w}) = -\sum_{t \in T} p_{G,w}(t) \log p_{G,w}(t)$$

Note that a uniform distribution over all possible test cases has the maximum entropy, thus maximizing the entropy objective function takes the generator distribution closer to a uniform distribution over all possible generations.

3.3.2 Targeting Valid Generations. Many common PBT examples have *preconditions* that define the set of valid inputs to the program under test. For instance, to test a program that inserts elements into a red-black tree, it is only useful to generate trees that satisfy the red-black tree invariant. To perform automated tuning for this purpose, we define the *specification objective function*.[6]

Definition 5 (specification objective function). *Given a generator G for test cases of type T and a validity condition $\phi : T \to \{0, 1\}$, the specification objective function is defined as*

$$\mathit{Specification}(w) := \log p_{G,w,\phi}(1) = \log \left(\sum_{\substack{t \in T \\ \phi(t)=1}} p_{G,w}(t) \right).$$

Intuitively, the specification objective function attempts to maximize the probability that a generated test case meets the validity condition ϕ.

3.3.3 Targeting Diverse, Valid Generations. The previous subsections discussed objective functions to target diversity and validity. However, these two objectives inherently conflict. Tuning for diversity incentivizes large terms, which are more likely to be diverse but less likely to be valid. Tuning for validity incentivizes trivially valid terms such as empty trees or lists. As a result, the common technique of combining objectives by taking their weighted sum is not effective here. To resolve this tension, we introduce the *specification entropy objective function*, which targets the entropy of the generator distribution *within* the space of valid test cases.

[6] This definition of the specification objective function is in accordance with Xu et al. [52].

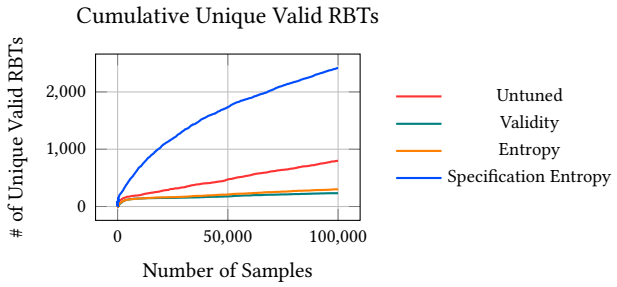


Fig. 5. Cumulative unique valid red-black trees throughout sampling, for our type-based RBT generator tuned for the different objective functions. We regularize weights as described in Section 6.3.

Definition 6 (specification entropy objective function). *Given a generator G and a validity condition $\phi : T \to \{0, 1\}$, the specification entropy objective function is defined as*

$$\mathit{SpecificationEntropy}(w) := -\sum_{\substack{t \in T \\ \phi(t)=1}} p_{G,w}(t) \log p_{G,w}(t).$$

This objective function aims to generate diverse test cases, except it disregards test cases that are invalid.

To illustrate these objectives, we took our type-based generator for color-labeled binary trees and tuned it for each of them. Once we had the tuned weights, we sampled 10^5 trees from the generator and computed the number of unique and valid RBTs we obtained over sampling. We show the results in Figure 5.

When we tuned this generator for the entropy objective function, we got diverse color-labeled binary trees but not many that were valid with respect to the RBT invariant. On the other hand, When we tuned it for validity we got valid test cases, but they are not diverse, as also shown in Figure 5. Indeed, that generator mostly produced red-black trees of height 0 in order to trivially satisfy the RBT invariant. Finally, Figure 5 shows that when tuned for specification entropy, the generator generates a much higher number of red-black trees that are both unique and valid, and it greatly outperforms the untuned version.

3.3.4 Targeting Diverse, Valid Generations with Respect to a Feature. In Section 3.2, we discussed how PBT users might wish to target a particular distribution over some feature of their generated test cases. What if they instead want to improve the diversity with respect to a feature? One can then tune the generator weights to maximize the entropy of the push forward of the generator distribution within the space of valid test cases. We define it formally as follows:

Definition 7 (feature specification entropy objective function). *Given a generator G, a function $g : T \to T'$, and a validity condition ϕ, the feature specification entropy objective function is defined as*

$$\mathit{FeatureSpecificationEntropy}(w) := -\sum_{\substack{t \in T \\ g(t)=t' \\ \phi(t)=1}} p_{G,w}(t) \log p_{G,w,g}(t').$$

Example 4. *Consider that G is a type-based STLC generator, and one wishes to tune its weights to produce well-typed terms that are of diverse types. Here, ϕ is the validity condition of well-typedness, and g is a function that takes as input an STLC term and outputs its type. Then we can tune the weights of G using feature specification entropy.*

$$
\begin{array}{rlll}
\textit{Types} & \tau & ::= & \text{Bool} \mid \tau_1 \times \tau_2 \\
\textit{Values} & v & ::= & T \mid F \mid (v, v) \\
\textit{Expressions} & aexp & ::= & x \mid v \\
 & e & ::= & aexp \mid \text{fst } aexp \mid \text{snd } aexp \mid \text{let } x = e \text{ in } e \mid \text{flip } q \\
 & & \mid & \text{if } aexp \text{ then } e \text{ else } e \\
\textit{Program} & p & ::= & e \\
\textit{Numeric Terms} & q & ::= & c \mid \theta
\end{array}
$$

Fig. 6. The syntax of Loaded Dice, which is a subset of Dice [26] except that unlike Dice we allow symbolic weights. The metavariable x ranges over variable names, c ranges over real numbers in the range $[0, 1]$, f ranges over function names, and θ ranges over variable names for symbolic weights.

$$
[\![v_1]\!]\,(v) \triangleq \big(\delta(v_1)\big)(v) \qquad [\![\texttt{fst}\ (v_1, v_2)]\!]\,(v) \triangleq \big(\delta(v_1)\big)(v) \qquad [\![\texttt{snd}\ (v_1, v_2)]\!]\,(v) \triangleq \big(\delta(v_2)\big)(v)
$$

$$
[\![\texttt{if}\ v_g\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ \texttt{e}_2]\!]\,(v) \triangleq \begin{cases} [\![\texttt{e}_1]\!]\,(v) & \text{if } v_g = \texttt{T} \\ [\![\texttt{e}_2]\!]\,(v) & \text{if } v_g = \texttt{F} \\ 0 & \text{otherwise} \end{cases} \qquad [\![\texttt{flip}\ c]\!]\,(v) \triangleq \begin{cases} c & \text{if } v = \texttt{T} \\ 1 - c & \text{if } v = \texttt{F} \\ 0 & \text{otherwise} \end{cases}
$$

$$
[\![\texttt{let}\ x = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2]\!]\,(v) \triangleq \sum_{v'} [\![\texttt{e}_1]\!]\,(v') \times [\![\texttt{e}_2[x \mapsto v']]\!]\,(v)
$$

Fig. 7. Semantics for Dice expressions. The function $\delta(v)$ is a probability distribution that assigns a probability of 1 to the value v and 0 to all other values.

Figure 4 shows how the distribution of the types of the generated STLC terms changes as the weights get tuned for both well-typedness and diversity of types. Recall from Section 2.4 that tuning decreased the proportion of terms that are ill-typed or of type `bool` from 91% to less than 1%.

4 Loaded Dice: A Language for Tunable Generators

The previous section described how we can map the intuition of PBT users to mathematical objectives. The users still need to write their generators whose weights can be tuned for these objective functions. For this purpose, we first describe our language Loaded Dice, where users can write their generators as probabilistic programs with symbolic weights. We then describe its implementation as an embedded DSL in Julia. Finally, we describe how users can add more tunable weights in their generators to increase their ability to be optimized for an objective function.

4.1 Syntax and Semantics

For automated generator tuning, we treat generators as probabilistic programs that represent distributions over test cases. For the purpose of writing generators, we describe Loaded Dice, an extension of the discrete probabilistic programming language Dice [26] with symbolic weights.

The syntax of Loaded Dice is given in Figure 6. Loaded Dice is a first-order functional language with support for booleans, tuples, and typical operations over these types.[7] It is augmented with

[7] Loaded Dice also provides support for *probabilistic conditioning*, but we omit it in Figure 6 because none of the generators require it.

the ability to create Bernoulli distributions via the `flip` syntax: the expression `flip` q represents a distribution that is true with probability q and false with probability $1 - q$. In Dice, arguments to `flip`s must be numeric constants; in Loaded Dice, they may also be *symbolic weights* denoted by metavariable θ. During generator tuning, it is these very symbolic weights that are tuned to optimize for an objective function. We describe the process of tuning in Section 6.

Loaded Dice inherits its semantics from Dice [26], replicated in Figure 7. For Dice, the semantic function $[\![\cdot]\!]$ maps expressions to probability distributions, where these probability distributions are functions from values to their probability mass. In order to support symbolic weights in Loaded Dice, we lift these semantics to the semantic function $[\![\cdot]\!]_L$, which maps expressions e with n symbolic weights and assignments $w \in [0,1]^n$ to the distribution that Dice semantics would result in if all symbolic weights were substituted by w. Formally,

$$[\![e]\!]_L(w) \triangleq [\![e[\theta \mapsto w]]\!].$$

The Loaded Dice semantics provide us a representation of the distribution in terms of the symbolic weights, which we require to learn weights as described in Section 6.

4.2 Implementation

Types	τ	::=	t | `Int` | `Bool` | `Tuple{`$\tau_1, \ldots$`}`
Statements	s	::=	**`@type`** $t = C_1(\tau_{11}, \ldots)$ "|" $C_2(\tau_{21}, \ldots)$ "|" $\ldots$
Numeric Terms	q	::=	c | θ
Expressions	e	::=	**`@match`** e $(C_1(x_{11}, \ldots) \to e_1,\ C_2(x_{21}, \ldots) \to e_2, \ldots)$
			| **`@dice if`** e_1 e_2 **`else`** e_3 **`end`** | `flip(`q`)` | $(e_1, \ldots)$
			| `freq(`$q_1 \Rightarrow e_1, \ldots$`)` | `backtrack(`$q_1 \Rightarrow e_1, \ldots$`)`

Fig. 8. Syntax for the library functions and macros that make up the Julia embedding of Loaded Dice. The metavariable x ranges over variable names, C ranges over constructor names, t ranges over type names, c ranges over numeric constants, and θ ranges over symbolic weights.

We implement Loaded Dice as an embedded domain-specific language (DSL) in Julia, and include extensions that make it easier to express the kinds of generators used by practitioners.

We show the syntax for the library functions and macros that make up the Julia embedding of Loaded Dice in Figure 8. As shown in earlier examples, this embedding includes the ability to declare algebraic data types and to pattern match on them. We provide the standard `freq` and `backtrack`[8] combinators for PBT [33]. There are also constructs that correspond one-to-one with Loaded Dice, such as **`@dice if`**, `flip`s, and tuples. These constructs in the embedded language form a library of functions and macros which are composed within a larger Julia program.

To be precise, this Julia program is not a generator, but a metaprogram whose execution produces a Loaded Dice program in the syntax shown in Figure 6. Thus, the functions and macros of the embedded DSL (Figure 8) are a library for constructing the data structure representing Loaded Dice programs. The rest of the program is "just Julia" – there is no special handling of Julia's language constructs, which are simply executed to produce the Loaded Dice program. This allows one to use arbitrary constructs (such as loops, side effects, etc.) to aid in constructing a generator, without them being materialized in the final Loaded Dice program.

[8] `backtrack` samples from a list of optional values, resampling without replacement upon sampling `None`.

In order to implement data structures such as integers and inductive types, we *bit-blast* [13, 20] them to representations in terms of Loaded Dice's tuples and booleans. For example, integers are binary-encoded as tuples of booleans [15]. To represent algebraic data types (e.g. lists and trees), we encode them as sum types and use an explicit discriminator and placeholder values for absent components of the sum. Specifically, the sum type $\tau_1 + \tau_2$ is encoded as the nested product type $\text{Bool} \times \tau_1 \times \tau_2$. Here, the boolean indicates whether the value is of type τ_1 or τ_2 and the other two components encode the value. This can be generalized to sums of arbitrary numbers of types by using an integer value as the discriminator. For example, for lists of length up to two, `[10;20]` is encoded as `(2,10,(2,20,(1,)))` and `[10]` is encoded as `(2,10,(1,0,(0,)))`, where `2` is the tag for `Cons`, `1` is the tag for `Nil`, and `0` is a placeholder value for absent arguments.

Lowering expressions to the core language is also straightforward. Pattern matching on values of an algebraic data type is lowered to the use of `if` expressions, as is standard. The `freq` and `backtrack` combinators are lowered to `if` and `flip`. For example, `freq([`$q_1 \Rightarrow e_1$`,`$q_2 \Rightarrow e_2$`,`$q_3 \Rightarrow e_3$`])` is lowered to **`if`** `flip(`$\frac{q_1}{q_1+q_2+q_3}$`)` **`then`** e_1 **`else if`** `flip(`$\frac{q_2}{q_2+q_3}$`)` **`then`** e_2 **`else`** e_3 **`end`**.

5 Constructing Tunable Generators

Using Loaded Dice, users can define the structure of a generator and leave the weights undetermined, to be optimized by automated generator tuning. However, the space of distributions that are possible to achieve by tuning the weights of the generator is limited by the generator's structure.[9] This, in turn, limits the extent to which Loaded Dice can optimize for an objective function.[10] We describe how generators can be written to be more "tunable," and how we can derive tunable generators from a type definition.

5.1 Adding Dependencies

We describe general techniques to make generators more amenable to automated tuning.

5.1.1 Parameterizing Weights by Function Arguments. Consider the generator in Figure 9a. The user may wish to tune it for a particular distribution over heights. However, the generators' distribution over heights only depends on one symbolic variable, $\theta_{\texttt{leaf}}$, which limits the extent to which tuning can optimize this generator.

A simple way to increase the expressivity of a generator is to add weights that depend on information already in scope. Concretely, rather than using $\theta_{\texttt{leaf}}$ in all invocations of `genTree`, we can select one of multiple weights based on the current value of the `size` parameter, as shown in Figure 9b. Let m denote the maximum size that this function is called with (5, in Figure 9a). Note that this generator now uses $m + 1$ symbolic weights instead of only two.

To add more symbolic weights in their generator, the user does not have to be limited by the preexisting structure of their generator. They can add additional function arguments to their generators for more symbolic weights. For example, in the RBT generator, the user can pass down the chosen color to each subcall, in order to parameterize the symbolic weights by the color of the parent node. This is shown in Figure 9c. Now, the generator consists of $2m + 1$ tunable weights.

5.1.2 Correlating Random Choices in the Generator. We described how we can increase the number of symbolic weights in a generator by parameterizing them over the function arguments. This technique, including the change shown in Figure 9c, has another effect: it correlates random choices in the generator that were previously independent. In particular, the generator in Figure 9a chooses

[9] Using the notation from Section 3, a generator G with n symbolic parameters exhibits the space of distributions represented by $\{p_{G,w} \mid w \in [0,1]^n\}$.

[10] This is the classic problem of *underfitting*, which is well-studied in the machine learning literature [40].

```
genColor() = @dice if flip(θ_red) R() else B() end

genTree(size) =
  @match size (
    0 → Leaf(),
    S(n) →
      @dice if flip(θ_leaf)
        Leaf()
      else
        Branch(genColor(), genTree(n), genNat(), genNat(), genTree(n))
      end)

G = genTree(5)
```

(a) An RBT generator following the same structure as QuickChick's type-based generators.

```
genTree(size) =
  @match size (
    0 → Leaf(),
    S(n) → (
      w = @match size (1 → θ_1, ..., m → θ_m);
      @dice if flip(w)
        Leaf()
      else
        Branch(genColor(), genTree(n), genNat(), genNat(), genTree(n))
      end))
```

(b) Adding weights that depend on `size` increases the distributions over height the generator can express.

```
genTree(size, parentColor) =
  @match size (
    0 → Leaf(),
    S(n) → (
      w = @match (size, parentColor) ((0,R()) → θ_0R, (0,B()) → θ_0B, ..., (m,B()) → θ_mB);
      @dice if flip(w)
        Leaf()
      else
        c = genColor()
        Branch(c, genTree(n, c), genNat(), genNat(), genTree(n, c))
      end))
```

(c) Adding a function parameter allows weights to depend on both `size` and the color of the parent call.

```
genTree(size, leaf) =
  @match size (
    0 → Leaf(),
    S(n) →
      @dice if leaf
        Leaf()
      else
        leftLeaf, rightLeaf = freq([ θ_1→(F,F), θ_2→(F,T), θ_3→(T,F), θ_4→(T,T) ]);
        Branch(genColor(), genTree(n, leftLeaf), genNat(), genNat(), genTree(n, rightLeaf))
      end)
```

(d) Restructuring the generator to frontload choices allows correlations between choices to be introduced.

Fig. 9. Modifications to a generator for RBT trees (not necessarily valid ones) to increase the space of distributions it can exhibit, written with syntactic sugar in Loaded Dice.

between different constructors, namely `Leaf` and `Branch`, independently of the color of the parent node. But that is no longer the case in Figure 9c, which parametrizes symbolic weights by the `parentColor`. Depending on the parent `color`, the probability of choosing `Leaf` changes.

The resulting dependency among random choices allows the generator to more closely fit an objective function. One can add more dependencies by *frontloading* random choices, allowing them to be made in tandem. For example, the RBT generator in Figure 9a makes independent choices for the constructors of the two children of a `Branch`. Instead, frontloading these choices can correlate them, as shown in Figure 9d. The `genTree` function in that figure chooses the constructors for the two children beforehand and passes that information as arguments to the subcalls.

5.2 Automatically Deriving Generators with Dependencies

QuickCheck provides its users the convenience of deriving simple generators for types automatically from their definitions. We observe that we can do the same for algebraic data types in Loaded Dice, but we can additionally use the ideas from above to make these generators more "tunable" by introducing additional dependencies.

API. To automatically derive generators with additional dependencies, we provide the metaprogramming [6] function `derive_generator`. It takes as input an algebraic data type definition in Loaded Dice, an integer representing the maximum size of generated values, and an integer specifying the *stack lookback length*. Given this information, `derive_generator` produces a generator for the given type that is parameterized by a size value and a portion of the execution context. Specifically, the tracked execution context is a representation of a suffix of the call stack, up to the specified lookback length. For example, for a binary tree data type, a stack lookback length of 2 indicates that the choice of a node should depend on the execution trace since the recursive call corresponding to the node two levels higher. The generator also frontloads choices so that they can be correlated, as shown in the previous subsection, so it also passes down choices to the appropriate places.

Implementation. At a high level, the function `derive_generator` constructs the desired generator in Loaded Dice by introducing new symbolic weights for each possible value of the dependencies, specifically the current size and portion of the call stack. To provide more intuition, we illustrate how `derive_generator` works using the example of our type-derived RBT generator in Figure 10, where the user has provided a maximum size of 5 and a lookback window length of 2.

The function `derive_generator` first defines a type representing a choice of constructor (Lines 5 and 6 of Figure 10). It then defines a sized generator with three arguments: `size`, `stack` and `chosenCtor`. The derived generator here first checks if `size` is zero, in which case it simply generates a `Leaf`. If `size` is not zero, it pattern matches on `chosenCtor`, which represents the choice of which constructor to use at this node in the tree, and then makes the random choices for the arguments to that constructor.

The code in Figure 10 uses a helper function that we have created called `freqDep`. Like the `freq` combinator, `freqDep` makes a weighted random choice from a list of values. But while the `freq` combinator specifies the weights directly, `freqDep` takes *dependencies* as an additional argument, and it introduces a different set of symbolic weights per value of the dependencies. In line 18 of Figure 10, we use `freqDep` to choose among the eight possible combinations of (left-subtree constructor, color, right-subtree constructor) for the `Branch` node being created, but with different symbolic weights per value of `deps`, which includes the current size and relevant portion of the

call stack. The `freqDep` function is implemented as a sequence of conditionals that branches on the possible values[11] of `deps` and introduces separate symbolic weights for each one.

A more general treatment of `derive_generator` can be found in Appendix A.

```
@type Color = Red | Black
@type Tree = Leaf | Branch(Tree,Color,Int,Tree)

# Types automatically generated to represent the choice of constructor
@type ColorC = RedC | BlackC
@type TreeC = LeafC | BranchC

genColorHelper(size, stack, chosenCtor) = # elided for brevity

function genTreeHelper(size, stack, chosenCtor)
  deps = (size, stack, chosenCtor)
  @match size (
    0 → Leaf,
    S(size') →
      @match chosenCtor (
        LeafC → Leaf,
        BranchC → begin
          argChoices = freqDep(deps, [
            # We use () as the dummy value for the choice of the constructors for
            # the int argument, as it is a builtin type.
            [LeafC,RedC,(),LeafC], [LeafC,RedC,(),BranchC],
            [LeafC,BlackC,(),LeafC], [LeafC,BlackC,(),BranchC],
            [BranchC,RedC,(),LeafC], [BranchC,RedC,(),BranchC],
            [BranchC,BlackC,(),LeafC], [BranchC,BlackC,(),BranchC]])
          Branch(
            # 2 is the configured stack lookback length.
            # 0, 1, and 2 number the program locations where recursive calls are made.
            genTreeHelper(size', firstN(2, cons(0, stack)), argChoices[0]),
            genColorHelper(size', firstN(2, cons(1, stack)), argChoices[1]),
            genIntDep(deps),
            genTreeHelper(size', firstN(2, cons(2, stack)), argChoices[3]))
        end))
end

function genTree()
  rootCtor = freqDep((), [LeafC, BranchC])
  genTreeHelper(5, [], rootCtor) # 5 is the configured initial size
end
```

Fig. 10. An automatically derived generator for red-black trees with dependencies (slightly simplified for presentation purposes).

6 Automatically Tuning a Loaded Dice Generator

Given an objective function and a generator with symbolic weights, the task of generator tuning is to determine the weights that maximize the objective function. To achieve this, we use the typical

[11] This set of possible values is a static value with respect to the Loaded Dice program, and is computed in the lowering of `freqDep`.

optimization algorithm, gradient descent, which requires computing gradients of the objective function to determine the direction[12] in which to update the weights.

The naïve methods for both probabilistic inference and computing gradients require enumerating all execution paths in a discrete probabilistic program. Dice scales *inference* by exploiting program structure; we leverage its existing compilation strategy to scale the computation of gradients. Still, given a method to compute gradients of Loaded Dice programs, entropy-based objectives involve all possible test cases, and enumerating the distribution is similarly intractable. We instead adapt REFINFORCE, a gradient estimation technique, to replace this enumeration with sampling. Finally, we discuss regularization techniques to avoid overfitting generator weights.

6.1 Probabilistic Inference and its Gradient

Tuning generators via gradient descent requires computing the gradients of the objective function with respect to the symbolic weights. Since the objectives depend on the generator distribution (as described in Section 3), we need probabilistic inference as a primitive.

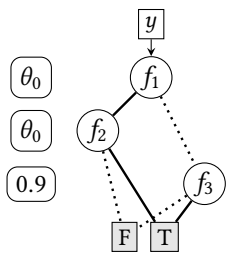
This poses the first key performance challenge: since probabilistic inference for arbitrary Dice programs is #P-hard in general, computing its gradients is also a #P-hard problem. Dice scales probabilistic inference by compiling programs to data structures that exploit program structure to compact the representation of distributions. In Loaded Dice, we leverage the very same compilation procedure to scale the computation of gradients.

6.1.1 Probabilistic Inference in Loaded Dice. Loaded Dice, as an extension of Dice, inherits its strategy for probabilistic inference. Dice compiles its probabilistic programs to ordered binary decision diagrams (OBDDs) and reduces the task of probabilistic inference to computations on the compiled OBDD. The task of probabilistic inference is #P-hard in general, but OBDDs exploit structure in probabilistic programs to reuse intermediate computations and scale probabilistic inference when possible.

An OBDD produced by Dice and Loaded Dice is a directed acyclic graph where each node corresponds to a boolean random variable with a *level* and is connected to two child nodes via a *high edge* (h) and a *low edge* (l). The high edge corresponds to the variable being true and the low edge corresponds to the variable being false. The two terminal nodes, `T` and `F`, don't have children. Each level corresponds to a `flip` in the program and its associated probability (which

```
let x = flip_1 θ0 in
let y = if x then flip_2 θ0
        else flip_3 0.9 in
y
```

(a) A Loaded Dice program.



(b) Compiled BDD representation.

$$
\begin{aligned}
pr(f_3) &= 0.9 \times 1 + (1 - 0.9) \times 0 = 0.9, \\
pr(f_2) &= \theta_0 \times 1 + (1 - \theta_0) \times 0 = \theta_0, \\
pr(f_1) &= \theta_0 \times pr(f_2) + (1 - \theta_0) \times pr(f_3) \\
&= \theta_0^2 + 0.9(1 - \theta_0).
\end{aligned}
$$

(c) The weighted model counting computation for node probabilities.

Fig. 11. A Loaded Dice program is first compiled to an OBDD which is then traversed via the procedure of weighted model counting (WMC). WMC generates an expression in terms of the symbolic weights, which can then be autodifferentiated to optimize via gradient descent.

[12]Typically, gradient descent updates the parameters in the *opposite* direction of the gradient, in order to *minimize* a loss function rather than to maximize an objective function. Thus we technically are performing gradient *ascent*.

may be symbolic in Loaded Dice). Given fixed values for each boolean variable, one can traverse the OBDD by starting from its root and following the edges corresponding to the value of the each variable to evaluate the program as `T` or `F`.

Once the program is compiled to an OBDD, the task of probabilistic inference is reduced to a bottom-up traversal on this graph. Each node n in an OBDD is associated with the probability $pr(n)$ of reaching the terminal `T` from that node. Our goal is to compute $pr(\texttt{root})$, which gets further used in the computation of the objective function. Now, $pr(\texttt{root})$ can be computed recursively in a single bottom-up pass of the OBDD using the following equations, where w is the weight associated with the level of the node and h and l are its high and low children, respectively [16, 19].

$$pr(\texttt{T}) = 1, \qquad pr(\texttt{F}) = 0 \qquad \text{(Base cases)}$$

$$pr(n) = w \cdot pr(h) + (1 - w) \cdot pr(l) \qquad \text{(Inductive Case)}$$

Note that the bottom-up traversal of an OBDD runs in time linear in the size of the OBDD [14]. This implies that the task of probabilistic inference also runs in time linear in the size of the OBDD. Thus, if one can get a small OBDD for a probabilistic program, one can also achieve efficient probabilistic inference for the program.

As an example, consider the Loaded Dice program in Figure 11a. Loaded Dice compiles this program to the OBDD in Figure 11b. Note that even though there are 8 possible instantiations of the coin flips in the program, the OBDD consists of only three nodes. For this OBDD, $pr(\text{root})$ can be computed as shown in Figure 11c, using the equations above.

6.1.2 Computing Gradients. Now, we describe how we can leverage the structure-exploiting properties of an OBDD to compute gradients efficiently. First, note that the expressions in Figure 11c are differentiable with respect to the symbolic weights. This is actually the case for exact probabilistic inference in Loaded Dice in general. As a result, probabilistic inference in Loaded Dice can support generator tuning via gradient descent.[13]

The question that remains is how one can actually compute these gradients efficiently. The standard method in machine learning libraries [1, 12, 43] is to compute gradients using *automatic differentiation* over a *computation graph*. The computation of these gradients scales linearly with the size of the computation graph. So if the computation graph is small, computing gradients is efficient. In fact, the OBDD that was earlier used for inference is exactly the computation graph we compute gradients for. All we need are the partial derivatives of $pr(n)$ for an OBDD node n, which compose via the chain rule of differentiation to compute the overall gradient.

$$\frac{\partial pr(n)}{\partial \theta} = (pr(h) - pr(l)) \qquad \frac{\partial pr(n)}{\partial pr(h)} = \theta \qquad \frac{\partial pr(n)}{\partial pr(l)} = 1 - \theta$$

Particularly, in Loaded Dice, we use the above equations to implement reverse-mode differentiation [5] over the compiled OBDDs.

To have a complete picture of the workflow, consider again the Loaded Dice program in Figure 11a. The Loaded Dice compiler first compiles it to an OBDD as shown in Figure 11b. Then, via the process of weighted model counting, the compiler produces code corresponding to the resulting probability distribution, equivalent to that shown in Figure 11c. Since objective functions are computed in terms of this resulting probability distribution, the generated code along with the code to compute the objective functions constitutes a symbolic loss function and can be autodifferentiated to obtain gradients for each weight in the Loaded Dice program. These gradients are then used to perform one update in the algorithm of gradient descent.

[13] Throughout this paper, we initialize weights to have uniform values.

6.2 Scaling Gradient Computation for Entropy-Based Objective Functions

Now that we can efficiently compute gradients of the objective function, we can use gradient descent to automatically tune the generators. But a key performance challenge still remains: for entropy-based objectives (Section 3.3), the objective functions themselves enumerate all possible test cases, as they are expectations with respect to the generator distribution. For example, consider the entropy objective function below, written as an expectation.

$$Entropy(\theta) = H(p_{G,\theta}) = -\mathbb{E}_{x\sim p_{G,\theta}(.)} \log p_{G,\theta}(x) = -\sum_{x\in X} p_{G,\theta}(x) \log p_{G,\theta}(x)$$

Computing exact gradients for this function requires differentiating inference for all possible test cases that the generator can produce, which is not amenable to scaling. To scale this computation, one may approximate the objective function via Monte Carlo sampling [45], as expectations can be approximated by averaging the inner expression over N samples. The resulting computation graph looks like Figure 12. However, to compute the gradients over this computation graph, one needs to differentiate through the sampling operation, but as a discrete operation, it is not differentiable.

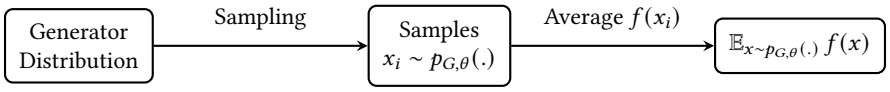


Fig. 12. The computation graph for approximating an expectation. To approximate $\mathbb{E}[f(x)]$, we can sample from the generator distribution and average $f(x)$ over the samples.

To resolve this issue, we adapt a well-known gradient estimation technique from the literature, REINFORCE [51], to our context of automated generator tuning. Specifically, for entropy based objective functions, we can estimate its gradient as follows. We include the derivation in Appendix C.

$$\nabla_\theta \mathop{\mathbb{E}}_{x\sim p_{G,\theta}(.)} [\log p_{G,\theta}(x)] = \mathop{\mathbb{E}}_{x\sim p_{G,\theta}(.)} [\log p_{G,\theta}(x) \nabla_\theta \log p_{G,\theta}(x)]$$

The equation above eliminates the need to differentiate through the sample operation. Instead it reformulates the gradient as another expectation, known as a *gradient estimator*, which can be directly computed from the samples by the same process as Figure 12. Note that in the computation of the inner expression for each sample, Loaded Dice is used to compute the contained gradient exactly.

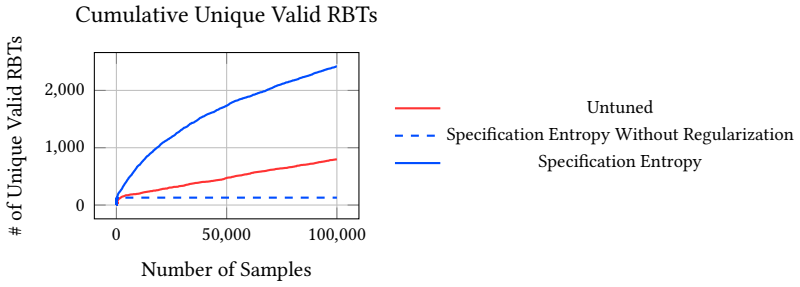


Fig. 13. Cumulative unique valid red-black trees throughout sampling, for our type-based RBT generator tuned for specification entropy, with and without regularization via bounded weights.

6.3 Regularization to Avoid Overfitting

While performing optimization using gradient descent, it is very common to run into the problem of overfitting. Automated generator tuning is no exception to this problem. Overfitting happens when the weights of the generator are over-optimized for the objective function. Particularly, when we tune generators for the specification entropy objective function, the generator avoids producing more diverse terms to avoid the penalty for producing an invalid term.

To avoid overfitting, regularization turns out to be an effective technique. Typical regularization techniques include adding a *penalty term* to the objective function or eliminating certain values for the parameters. For effective generator tuning to avoid overfitting, we employ the latter and bound the weights in the generators between $[0.1, 0.9]$.

To demonstrate the effectiveness of using a regularization technique, we tune a type-based generator for red-black trees for the specification entropy objective function with and without regularization. Once the generators are tuned, we sample 10^5 trees using these generators and record the number of unique and valid RBTs we obtain. It is clear from the results shown in Figure 13 that tuning with regularization allow the generators to achieve a much higher number of unique valid red-black trees as we obtain more samples.

7 Evaluation: Bug-Finding on Etna Benchmarks

Previous sections, and in particular Section 2, have already demonstrated that our approach gives developers better control over their generators' distributions. The experiments in Figure 2, Figure 4, Figure 5, and Figure 13 show that tuned generators are successfully optimized for various objective functions.

In this section, we show that the better control we afford developers is actually useful for the core purpose of PBT. In other words, we ask:

How effective is our approach at improving bug-finding performance?

To investigate this question, we implemented Loaded Dice as an embedded domain-specific language in Julia [8] and used it to test various case studies from the Etna benchmark suite [47]. Etna was specifically developed to evaluate PBT tools on how quickly they find bugs (pre-placed by the benchmark authors) in example programs.

7.1 The Testing Workloads

We evaluate our approach on three of the four Rocq workloads from the Etna benchmark suite. The three workloads are designed to evaluate different PBT approaches to test programs that take as inputs binary search trees (BST), red-black trees, and terms of the simply-typed lambda calculus (STLC). Each of these workloads has a set of generators as well as a set of "tasks," or bugs intentionally planted in the programs. We use our approach to tune generators based on the following strategies for these three workloads:

- **STLC, BST, and RBT Type-Based Generators**: We automatically derived type-based generators with additional dependencies for these types, as described in Section 5.2, with a stack lookback length of 2. We then tuned these generators for diversity and validity via specification entropy using our approach. We compare these tuned generators against the untuned[14] versions as well as the original generators.
- **STLC Bespoke Generator**: We first adapted Etna's bespoke STLC generator, a handwritten generator for STLC terms that uses backtracking to always generate well-typed terms, to Loaded Dice. We fixed initial sizes and parameterized weights by the size argument of the

[14] As with the rest of this paper, an "untuned" generator is one in which each random choice is uniformly distributed.

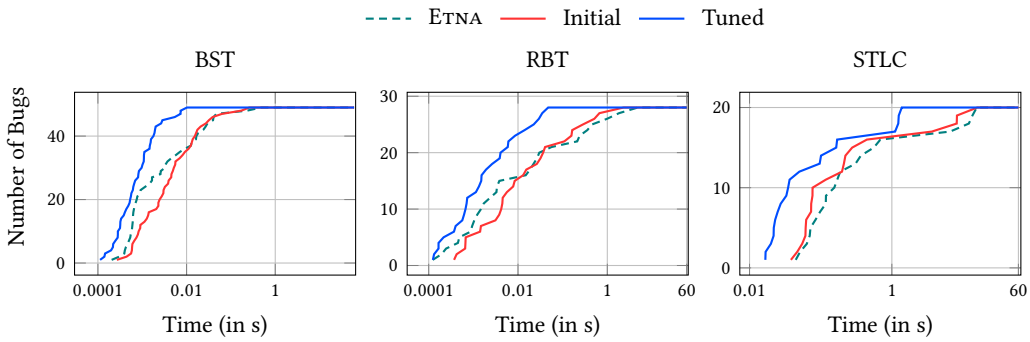
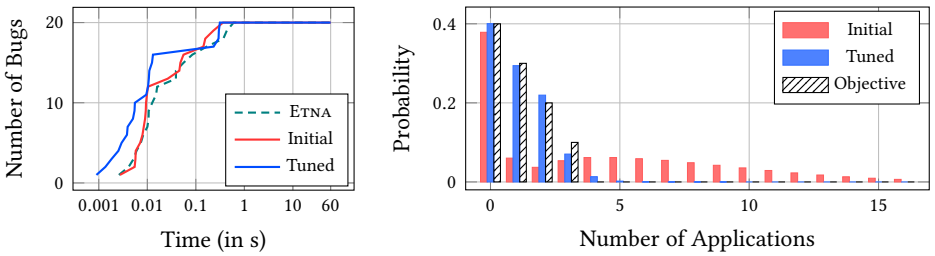


Fig. 14. Time in seconds vs. number of bugs found (higher is better) across workloads and generator strategies. The x-axes use log scaling.



(a) Time in seconds vs. number of bugs found (higher is better). The x-axis uses log scaling.

(b) Distribution of the number of applications. Truncated for space; all omitted values (max: 31) have probability less than 0.005.

Fig. 15. The results from tuning a bespoke STLC generator to leverage insight from Shi et al. [47].

current recursive call (Section 5.1). The generator is shown in Appendix B. We then tuned it to leverage existing insight by the authors of Etna. In particular, they make the observation that larger generations can be empirically detrimental for bug-finding (in the face of conventional PBT wisdom). In accordance with this observation, we tuned the STLC Bespoke generator for a target distribution over the number of syntactic function applications (App constructors), i.e. $\{0 \rightarrow 40\%, 1 \rightarrow 30\%, 2 \rightarrow 20\%, 3 \rightarrow 10\%\}$. We compare the tuned generator against the untuned one as well as the original bespoke generator.

7.2 Methodology

We evaluated each of the above generators for the coverage and speed of bug-finding. We report the time (median over 11 trials) it takes each of these generators to find the bugs in their respective workloads in Figure 14 and Figure 15. We report a timeout for a particular bug at 60 seconds. We also report the time it took to tune the weights of these generators in Appendix D.

7.3 Results

We found that our approach significantly improved bug-finding speed in all four of our benchmarks, in comparison to both the untuned versions and the generators from Etna. For the type-based stategy, tuning for specification entropy increased the bug-finding speed of our generators by

3.1–7.4× over the untuned versions and the QuickChick generators (Table 1). Figure 14 additionally shows that the number of bugs found by the tuned generator is greater or equal to the others at all times. For the bespoke STLC generator, tuning for a target distribution over the number of applications increased bug finding speed by at least 1.9× over the untuned version and the original version in Etna (Table 1, Figure 15a). Additionally, Figure 15b validates that the distribution of applications did change as intended by tuning.

Training time for our tuned generators ranged from three to eight minutes. One may notice that minutes of training time is significant compared to seconds to find bugs. However, this is a one-time cost, whereas generators are frequently run in continuous integration, sometimes as frequently as every code change. Thus, the training cost amortizes over multiple testing runs. We provide further detail into the training costs for our tuned generators in Appendix D.

Table 1. Relative bug-finding speed of tuned generators over the original Etna generators and the untuned generators with additional parameters. We report the geometric mean of the speedup relative to each baseline for all tasks in the workload. Timing data are from the same evaluation runs as Figure 14 and Figure 15a.

	Relative Bug-Finding Speed	
Generator & Workload	Etna → Tuned	Initial → Tuned
BST Type-Based	3.5×	5.4×
RBT Type-Based	5.8×	7.4×
STLC Type-Based	4.3×	3.1×
STLC Bespoke	2.3×	1.9×

7.4 Internal Evaluation: Cause of Bug-Finding Speedup

While these results show that tuning significantly improves bug-finding performance, it is important to understand the cause of these speedups. We hypothesize that specification entropy improves bug-finding by increasing the number of valid and unique samples, and that tuning the STLC Bespoke generator for smaller terms increases bug-finding speed because of the increased generation speed. To confirm this hypothesis, we measure the change in generation speed and the change in number of unique and valid samples and report in Table 2.

7.4.1 Generation Speed. Tuning generators for specification entropy increased their bug-finding speed (3.1–7.4×) significantly more than their generation speed (0.5–1.3×), indicating that changes in generation speed were *not* the primary cause of their speedup. Conversely, tuning the STLC bespoke generator for smaller generations increased its generation speed (4.4×) more than its bug-finding speed (1.9×), also as expected.

We also observe that adapting Etna generators to the initial Loaded Dice generators changed generation speed (0.6–1.3×) significantly less than the bug-finding speedup of using tuned Loaded Dice generators instead of Etna generators (2.3–5.8×). This indicates that the primary advantage of the tuned generators was caused by the tuning itself.

7.4.2 Valid and Unique Samples. We see in Table 2 that tuning for specification entropy significantly increases the number of valid and unique samples (matching the results of our prior experiment in Figure 5). In contrast, tuning the STLC Bespoke generator for smaller generations *decreases* the number of valid and unique samples. This highlights the tradeoff to be made depending on the tuning objective — the virtue of tuning is that it allows one to choose that tradeoff.

We also provide validity rates of each generator, which consistently increases with tuning for specification entropy, in Appendix E.

Table 2. Relative generation speeds, and number of unique, valid samples out of 100,000 samples, for generators evaluated in Etna.

	Relative Generation Speed		Unique, Valid Samples	
Generator & Workload	Etna → Initial	Initial → Tuned	Initial	Tuned
BST Type-Based	0.6×	0.9×	2,592	14,387
RBT Type-Based	0.6×	1.3×	767	2,394
STLC Type-Based	0.8×	0.5×	771	11,181
STLC Bespoke	1.3×	4.4×	55,322	10,589

8 Discussion

The previous section demonstrates empirically that our approach for automatic generator tuning enables faster bug finding. In this section, we discuss the larger context in which our approach fits.

8.1 Flat and Statically-Bounded Generators

Probabilistic programming languages such as Loaded Dice restrict the language they support in order to make the task of exact inference tractable. In contrast, typical generator languages such as QuickCheck are much richer. As a result, not all generators are expressible in Loaded Dice. Specifically, Loaded Dice only supports first-order (flat) generators that are statically bounded.

However, it turns out that many practical generators naturally tend to be expressible in first-order. All of the generators that are automatically produced from type information by tools like DRaGeN [39] and generic-random [36] fit this criteria, as do all specification-based generators derived from Rocq inductive relations in QuickChick [32, 41]. Handwritten generators tend to fit this pattern too: all generators used in the benchmark suite provided by the Etna evaluation tool are flat, including complex and highly-tuned generators for well-typed programs in the simply-typed lambda calculus and System $F_{<:}$.

8.2 Adaptive Sizing

Some generator frameworks, such as QuickCheck, have a notion of *sized* generators. These are functions that output a generator given a `size` parameter. This allows the framework to sample test cases by progressively increasing the size of the generator. This is called *adaptive sizing*.

Since Loaded Dice can only express statically-bounded generators, to use our approach the initial size has to be specified. In Section 7, we fix the initial size of our generators at 4, 4, and 5 for BST, RBT and STLC workloads respectively and find that generators tuned with our approach can outperform adaptively-sized generators anyway (generators referred to as "Etna" in Table 1, Figure 14, and Figure 15). Loaded Dice also supports modeling adaptive initial sizes as a fixed distribution, resulting in a generator that can be tuned as a proxy for the adaptively-sized version.

9 Related Work

We report on related work for both generator tuning and probabilistic programming.

9.1 Tuning Generators

There are a number of existing approaches to generator tuning for PBT.

Manual tuning. Perhaps the simplest solution to generator tuning is to provide knobs for the user to do it manually. Indeed, this was part of the motivation for the original QuickCheck [18] and QuickChick [42] generator languages. This approach is simple, but it has the limitations we

discuss in the introduction: tuning a generator by hand requires significant experimentation, since it's hard to know the overall distribution that will be produced by a particular collection of local weights. Loaded Dice offers an automatic approach.

Besides language-based approaches, there are auxiliary tools aimed at easing the tuning process. In particular, Tyche [23] is a visual user interface that provides insights into a generator's current distribution. This approach is complementary to Loaded Dice, as it could be used to confirm that an objective had the intended effect, or to visually compare objectives. In turn, automatic tuning could enhance such interfaces, e.g., by allowing the user to click-and-drag to adjust distributions.

Online tuning. One alternative to manual tuning is tuning during the testing process. The Target system [37] uses hill-climbing and simulated annealing during generation to maximize an objective. RLCheck [44] tunes generators with reinforcement learning, seeking out diverse and valid inputs. Choice Gradient Sampling [22] tunes online by manipulating the generator representation itself. ISLa [48] uses an SMT solver during generation to improve the chance of finding valid inputs.

Online techniques allow the generation process to target objectives over time, but it is hard to predict the impact that they will have on the ultimate distribution of generated test cases. By contrast, our approach gives direct control over that final distribution. Furthermore, online approaches usually do a significant amount of work during generation, leading to relatively slow sampling speeds. A recent study by Goldstein et al. [21] suggests that some users of PBT run their properties in a very tight loop, testing their properties as often as every time they save their code. In these cases, online tuning, whether that means running a learning algorithm or calling an SMT solver, may waste precious time that could be used finding bugs.

Tuning by construction. Some existing techniques automatically derive PBT generators from data types or inductive specifications. For example, DRaGeN [39] computes weights based on insights from the literature on branching processes, aiming to uniformly distribute data constructors. DRaGeN is faster than Loaded Dice at deriving and tuning generators, but is hard-coded for a particular distributional goal and does not work for user-defined generators. Similarly, Lampropoulos et al. [32] derive effective generators from inductive relations in the Rocq theorem prover. This process produces specification-satisfying generators by construction, but requires more up-front effort from the user and does not support distributional requirements beyond validity.

9.2 Differentiating Discrete PPLs

Automatic differentiation has been used in probabilistic programming systems before, but only to compute the gradients of likelihood functions in order to guide the probabilistic inference algorithms. At a high level, the key novelty of Loaded Dice is that it uses automatic differentiation to compute *exact* gradients of probabilistic inference itself.

Gradient-Based Inference Algorithms. Probabilistic inference for arbitrary probabilistic programs is hard. This has led to inference algorithms informed by gradients of the likelihood functions. Notably, Hamiltonian Monte Carlo [7] chooses the next sample using the gradient of the likelihood of the current sample. Variational inference [9, 10] utilizes gradients of the likelihood to approximate the posterior distribution via a family of closed form distributions. On the other hand, Loaded Dice performs discrete probabilistic inference and then computes the exact gradient with respect to the weights. Both HMC and variational inference are limited to continuous probabilistic programs, whereas Loaded Dice provides support for discrete programs, which is necessary for the application to PBT for data structures such as lists and trees.

Alternate Formulations of Parameter Tuning. We cast the problem of generator tuning as one of gradient descent relative to an objective function. An alternative would be to cast the problem as a

form of Bayesian inference, which learns a posterior distribution for a model's parameters given a set of observations about the data generated by the model. In this formulation, the desired objective function would be modeled as a set of observations. However, this approach would require casting our entropy objective as an observation, but entropy is not an event that can be observed but is rather a property of the entire distribution. In principle, one can cast our target distribution objective as an observable event. But again, this would involve reasoning about the generator distribution in its entirety rather than a single value the generator can produce. Thus, computing the posterior distribution of the generator weights in this setting would be highly intractable.

Another alternate formulation would be to treat our objective functions as likelihoods and attempt to compute the maximum-a-posteriori (MAP) estimation for the parameters in the generator under consideration. Computing the MAP estimation is an optimization problem that can be computed via gradient-based approaches, and so that would be equivalent to what our approach accomplishes [40].

Gradient Estimators for Discrete Probabilistic Programs. Recent work such as ADEV [34] and `StochasticAD.jl` [4] also compute gradients of probabilistic inference but via sampling. They propose program transformations to produce gradient estimators for probabilistic programs. These works offer different variance and scaling tradeoffs: while Loaded Dice computes exact, zero-variance and zero-bias gradients, they employ Monte Carlo sampling to approximate these gradients.

Learning in Probabilistic Logic Programming. Loaded Dice compiles programs to binary decision diagrams and differentiates through them to learn weights. Similar functionality is found in probabilistic logic programming where systems like DeepProblog and Scallop [25, 35, 38] allow users to provide first-order logical specifications with weights that can be learned as outputs of neural networks. Loaded Dice differs from these approaches as it supports more traditional programming constructs; we also provide specific learning objectives and associated algorithms to improve PBT.

10 Conclusion and Future Work

In this paper, we presented a novel framework for automatically tuning PBT generators. We described how different intuitions of users about their generator distributions can be mapped to objective functions. We presented a new PPL, Loaded Dice, to express generators with support for symbolic weights and parameter learning. We also described how automated generator tuning can be made feasible and demonstrated its benfits in enabling PBT generators to find bugs faster.

In the future, we hope to extend Loaded Dice to provide support beyond flat and statically-bounded generators. In particular, common PBT frameworks support nested generators that induce distributions over distributions. We hope to reduce them to flat generators via *defunctionalization*. We also hope to provide support for adaptive sizing. Finally, we wish to explore how automated generated tuning, by allowing distributions to be specified declaratively rather than operationally, can enable more user-friendly interfaces and APIs for property-based testing.

11 Data-Availability Statement

The artifact for this paper consists of the implementation of Loaded Dice as an embedding in Julia and code to reproduce experiments and plots in Sections 2, 3, 6 and 7. It is available on Zenodo [50]. Loaded Dice is also available as an open-source repository on GitHub at https://github.com/Tractables/Alea.jl/tree/loaded-dice.

Acknowledgments

We would like to thank Leonidas Lampropoulos for his support and guidance as well as Steven Holtzen and Zilei Shao for useful technical discussions. This work is supported in part by the National Science Foundation under grants CCF-2220891, *SHF: Medium: Usable Property-Based*

Testing, NSF #2402449 and IIS1943641, the Victor Basili Postdoctoral Fellowship at the University of Maryland, DARPA ANSR, CODORD, and SAFRON programs under awards FA8750-23-2-0004, HR00112590089, and HR00112530141, and gifts from Adobe Research, Cisco Research, and Amazon. Approved for public release; distribution is unlimited.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://doi.org/10.5281/zenodo.16852354 Software available from tensorflow.org.

[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] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Christopher Rackauckas. 2022. Automatic Differentiation of Programs with Discrete Randomness. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 10435–10447. https://proceedings.neurips.cc/paper_files/paper/2022/file/43d8e5fc816c692f342493331d5e98fc-Paper-Conference.pdf

[5] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. 2000. Automatic differentiation of algorithms. *J. Comput. Appl. Math.* 124, 1 (2000), 171–190. https://doi.org/10.1016/S0377-0427(00)00422-2 Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[6] Alan Bawden. 1999. Quasiquotation in Lisp. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, Olivier Danvy (Ed.). University of Aarhus, 4–12.

[7] Michael Betancourt. 2018. A Conceptual Introduction to Hamiltonian Monte Carlo. arXiv:1701.02434 [stat.ME] https://arxiv.org/abs/1701.02434

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM Review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[9] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

[10] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112, 518 (April 2017), 859–877. https://doi.org/10.1080/01621459.2017.1285773

[11] Avrim Blum, John Hopcroft, and Ravindran Kannan. 2020. *Foundations of Data Science.* Cambridge University Press. https://doi.org/10.1017/9781108755528

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/jax-ml/jax

[13] Roberto Bruttomesso and Natasha Sharygina. 2009. A scalable decision procedure for fixed-width bit-vectors. In *Proceedings of the 2009 International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '09)*. Association for Computing Machinery, New York, NY, USA, 13–20. https://doi.org/10.1145/1687399.1687403

[14] Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[15] William X. Cao, Poorva Garg, Ryan Tjoa, Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2023. Scaling integer arithmetic in probabilistic programs. In *Proceedings of the Thirty-Ninth Conference on Uncertainty in Artificial Intelligence (Proceedings of Machine Learning Research, Vol. 216)*, Robin J. Evans and Ilya Shpitser (Eds.). PMLR, 260–270. https://proceedings.mlr.press/v216/cao23b.html

[16] Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172, 6 (2008), 772–799. https://doi.org/10.1016/j.artint.2007.11.002

[17] 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

[18] 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

[19] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming* 15, 3 (2015), 358–401. https://doi.org/10.1017/S1471068414000076

[20] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2024. Bit Blasting Probabilistic Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 182 (June 2024), 24 pages. https://doi.org/10.1145/3656412

[21] 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

[22] 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

[23] Harrison Goldstein, Jeffrey Tao, Zac Hatfield-Dodds, Benjamin C. Pierce, and Andrew Head. 2024. Tyche: Making Sense of PBT Effectiveness. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3654777.3676407

[24] Leo J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. 8–21. https://doi.org/10.1109/SFCS.1978.3

[25] Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. 2008. Parameter learning in probabilistic databases: a least squares approach. In *Proceedings of the 2008th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I* (Antwerp, Belgium) *(ECMLPKDD'08)*. Springer-Verlag, Berlin, Heidelberg, 473–488. https://doi.org/10.1007/978-3-540-87479-9_49

[26] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 140 (nov 2020), 31 pages. https://doi.org/10.1145/3428208

[27] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer International Publishing, Cham, 169–186. https://doi.org/10.1007/978-3-319-30936-1_9

[28] John Hughes. 2019. How to Specify It!. In *20th International Symposium on Trends in Functional Programming*. https://doi.org/10.1007/978-3-030-47147-7_4

[29] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 135–145. https://doi.org/10.1109/ICST.2016.37

[30] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79 – 86. https://doi.org/10.1214/aoms/1177729694

[31] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, (POPL)*. https://doi.org/10.1145/3009837.3009868

[32] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating good generators for inductive relations. *Proc. ACM Program. Lang.* 2, POPL, Article 45 (dec 2017), 30 pages. https://doi.org/10.1145/3158133

[33] Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*. Electronic textbook. http://www.cis.upenn.edu/~bcpierce/sf

[34] Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 5 (jan 2023), 33 pages. https://doi.org/10.1145/3571198

[35] Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. arXiv:2304.04812 [cs.PL] https://arxiv.org/abs/2304.04812

[36] Li-yao Xia. 2018. *A quick tour of generic-random*. https://hackage.haskell.org/package/generic-random-1.5.0.0/docs/Generic-Random.html

[37] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

[38] Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. DeepProbLog: Neural Probabilistic Logic Programming. arXiv:1805.10872 [cs.AI] https://arxiv.org/abs/1805.10872

[39] Agustín Mista and Alejandro Russo. 2021. Deriving compositional random generators. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/3412932.3412943

[40] Kevin P. Murphy. 2022. *Probabilistic Machine Learning: An introduction.* MIT Press. http://probml.github.io/book1

[41] 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

[42] 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

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703

[44] 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

[45] Christian P. Robert and George Casella. 2005. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-1-4757-4145-2

[46] Claude Elwood Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[47] Jessica Shi, Alperen Keles, Harrison Goldstein, Benjamin C. Pierce, and Leonidas Lampropoulos. 2023. Etna: An Evaluation Platform for Property-Based Testing (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 218 (aug 2023), 17 pages. https://doi.org/10.1145/3607860

[48] 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

[49] Donald Stewart, Koen Claessen, Nick Smallbone, and Simon Marlow. 2024. Test.QuickCheck — Hackage.Haskell.Org.

[50] Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd Millstein, Benjamin Pierce, and Guy Van den Broeck. 2025. *Artifact for: Tuning Random Generators: Property- Based Testing as Probabilistic Programming.* https://doi.org/10.5281/zenodo.16868014

[51] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (1992), 229–256. https://doi.org/10.1007/BF00992696

[52] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. 2018. A Semantic Loss Function for Deep Learning with Symbolic Knowledge. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5502–5511. https://proceedings.mlr.press/v80/xu18h.html

Received 2025-03-25; accepted 2025-08-12