[DRAFT] 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 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.

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

1 Introduction

Property-based testing (PBT) is a powerful [2, 3, 26] and widely-studied [20, 24, 25] software testing technique that validates a system under test with respect to an executable specification by 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 [15, 20].

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 and on methods for automatically deriving generators from data type definitions [32] or inductive relations [28].

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

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 an embedding of the Loaded Dice PPL (Section 4) in Julia; `@match` is a Julia macro defined by Loaded Dice to implement pattern matching.

```
genList(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 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 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. Indeed, recent work on PBT usability [21] 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 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* [41], 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 [23], 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 [14], 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 [46], 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 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 5).
- 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 6).

2 Overview

In this section, we give an overview of our approach, focusing in particular on how it can provide benefit for PBT users.

2.1 The Basics of Generator Tuning

Since the beginning of PBT, in QuickCheck [16], 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:

```
1 target = ['a' ⇒ 0.2, 'b' ⇒ 0.2, 'c' ⇒ 0.2, 'd' ⇒ 0.2, 'e' ⇒ 0.2]
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[1] that are proportional to the following mapping,

```
[θ1 ⇒ 1, θ2 ⇒ 2, θ3 ⇒ 2, θ4 ⇒ 1,
 θ5 ⇒ 1, θ6 ⇒ 1, θ7 ⇒ 2, θ8 ⇒ 2]
```

which achieves the desired distribution.

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 a generator like the one in Figure 1, which generates random color-labeled binary trees (some subset of these trees will be valid red-black trees [45], but this generator does not ensure that invariant). The function `genTree` takes a maximum tree `size` and then generates trees up to that

[1]Weights can be configured to be initialized randomly or to particular values. Throughout this paper, we initialize weights to have uniform values.

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

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

genTree(size) =
  @match size (
    O → 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 binary trees (not necessarily valid red-black trees), using symbolic weights. The macros `@type` and `@match`, provided by the Loaded Dice embedding in Julia, extend Loaded Dice with algebraic datatypes.

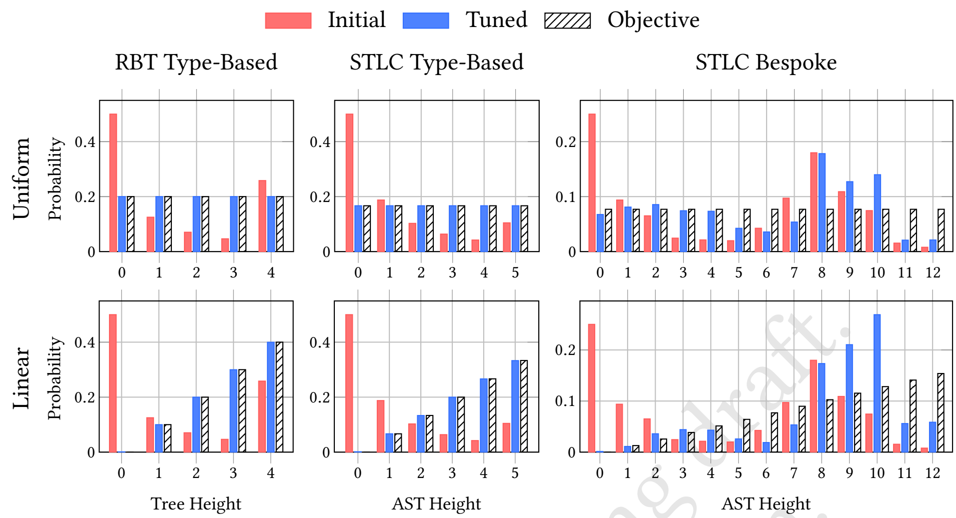


Fig. 2. Tuning the distributions of heights of generated values to be uniform or linear for several generators. Each column is a different generator; the top row trains the generators to have a uniform distribution over heights, and the bottom row trains the generators to have a linear distribution over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

size. If `size` is non-zero, `genTree` makes a random choice: with probability $\theta_{\texttt{leaf}}$ it generates a leaf, and with probability $1 - \theta_{\texttt{leaf}}$ 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.

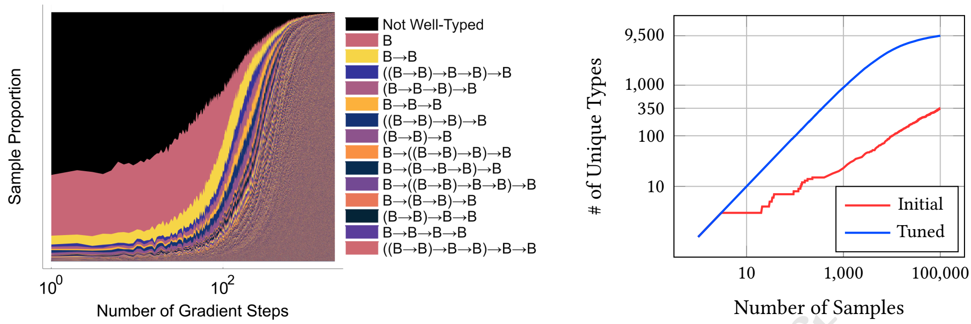
Starting from this generator, a developer might have some ideas for what they would like the distribution of trees to look like. For example, they may tune the generator to produce relatively few very small trees (height 1 and 2) and proportionally more large trees (height 4 and 5). They can specify 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 are often better at finding bugs.[2]

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 A). For each, 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[3] versions (in red).

[2]As observed by Shi et al. [42] and shown in Section 6, it can also be beneficial to tune for *smaller* inputs.
[3]Throughout this paper, an "untuned" generator is one in which each random choice is uniformly distributed.
[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.



(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. 3. 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."

Sections 3, 4, and 5 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 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 language called Loaded Dice with two significant differences. First, Loaded Dice has *symbolic weights* (the θs in the generators above) that 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 not only compute the distribution of a generator, we can also compute how the symbolic weights should be changed to maximize a given objective function. Together, these imply an algorithm for choosing values for symbolic weights by gradient descent: the system repeatedly computes the gradient of the generator with respect to an 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 the generator as close as possible to the desired distribution [10].

This explanation glosses over critical details about how we make this process efficient and effective, but users of our approach should be able to get started with this level of understanding.

2.3 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 a number of 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* in the generated values. For generators producing unconstrained values, this can be an ideal 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 define a notion of *specification entropy*, which attempts to simultaneously optimize entropy and validity, for this purpose. Figure 3 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 3a visualizes how the distribution over types changes over time as we tune the weights; Figure 3b 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, no single type comprises 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 [19]. 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 [21, 29, 44], 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 [27] 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.*

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

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 6, we use the target objective function to leverage insight from Shi et al. [42] 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 [41] 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 $p_{G,w}$.*

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

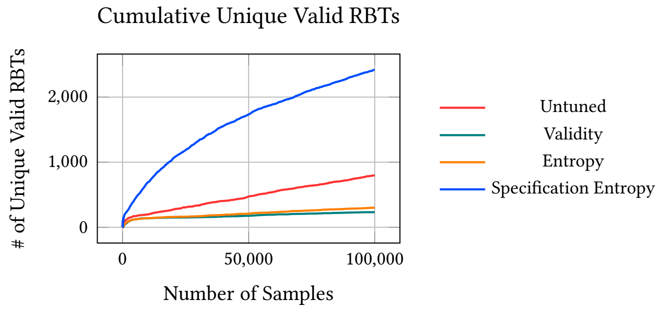


Fig. 4. Cumulative unique valid red-black trees throughout sampling, for our type-based RBT generator tuned for the different objective functions. We regularize weights as described in Section 5.3.

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.

To test the effectiveness the entropy objective function, as well as the other objective functions defined below, 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 4.

When we tuned this generator for the entropy objective function, we got diverse color-labeled binary trees but not ones that were valid with respect to the RBT invariant. As a result, this generator produces fewer *valid* RBTs that it would have had it not been tuned at all! This is evident in Figure 4.

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.*[5]

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. When we tune our type-based generator for this objective function, we get valid test cases, but they are not diverse, as shown in Figure 4. The generator mostly produced red-black trees of height 0 in order to trivially satisfy the constraint.

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

[5]This definition of the specification objective function is in accordance with Xu et al. [47].

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.

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*

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

This objective function aims to generate diverse test cases, except it disregards test cases that are invalid. Figure 4 shows that when tuned for specification entropy, the generator generates a much higher number of unique valid red-black trees and 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*

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

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.*

Figure 3 shows how the distribution of the types of the generated STLC terms changes as the weights get tuned for feature specification entropy. Recall from Section 2.3 that tuning decreased the proportion of terms that are ill-typed or of type `bool` from 91% to less than 1% of all generations.

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 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 a discrete probabilistic programming language Dice [23] with symbolic weights.

The core syntax of Loaded Dice is given in Figure 5. Loaded Dice is a first-order functional language with support for booleans, tuples, and typical operations over these types.[6] It is augmented

[6] Loaded Dice also provides support for *probabilistic conditioning*, but we omit it in Figure 5 because none of the generators require it.

Types	τ	::=	$\text{Bool} \mid \tau_1 \times \tau_2$	
Values	v	::=	$T \mid F \mid (v, v)$	
Expressions	$aexp$	::=	$x \mid v$	
	e	::=	$aexp \mid \text{fst } aexp \mid \text{snd } aexp \mid \text{let } x = e \text{ in } e \mid \text{flip } q$	
		$\mid$	$\text{if } aexp \text{ then } e \text{ else } e \mid f(aexp)$	
Functions	$func$	::=	$\text{fun } f(x : \tau) : \tau\{e\}$	
Program	p	::=	$e \mid func\ p$	
Numeric Terms	q	::=	$c \mid \theta$	(Symbolic weights θ only in Loaded Dice)

Fig. 5. The syntax of Dice [23] and Loaded Dice. 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. In Dice, symbolic weights are not allowed.

with 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 `flips` 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 5.

Loaded Dice inherits the semantics from Dice [23], replicated in Figure 6. 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 5.

4.1.1 Extensions: Our implementation includes various extensions to Loaded Dice that make it easier to express the kinds of generators used by practitioners. In addition to this core syntax, the language is extended with syntactic sugar for logical operations ($\wedge$, $\vee$, $\neg$, etc.), integers, and algebraic data types (e.g. lists and trees). To represent distributions over integers, we use the binary

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

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

$$[\![\mathtt{let}\ x = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2]\!]\,(v) \triangleq \sum_{v'} [\![\mathtt{e}_1]\!]\,(v') \times [\![\mathtt{e}_2[x \mapsto v']]\!]\,(v) \qquad [\![f(v_1)]\!]\,(v) \triangleq \Big(\big(T(f)\big)(v_1)\Big)(v)$$

Fig. 6. 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. The implicit context T maps function names to their semantics.

encoding represented using a vector of booleans [13, 18]. To express data structures like lists and trees, Loaded Dice encodes sum types $\tau_1 + \tau_2$ as nested product types 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 elements actually encode the value.

The functions in Loaded Dice are nonrecursive, but our implementation supports statically-bounded recursion as well as bounded loops, which we use freely in examples. While generators over inductive types are often naturally recursive, they typically use a size parameter to bound the maximum number of recursive calls and hence are expressible in our implementation of Loaded Dice. We discuss this in detail in Section 7.1.

Finally, our implementation provides library functions that match common idioms from PBT. For example, we provide the `freq` and `backtrack` [29] combinators, which can be implemented using `if` and `flip`.

4.2 Writing Generators in Loaded Dice

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.[7] This, in turn, limits the extent to which Loaded Dice can optimize for an objective function.[8] Here, we describe ways users can write their generators to make them more amenable for automated generator tuning.

4.2.1 Parameterizing Weights by Function Arguments. Consider the generator in Figure 7a. 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 7b. Let m denote the maximum size that this function is called with (5, in Figure 7a). 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 7c. Now, the generator consists of $(2 \times m + 1)$ tunable weights.

4.2.2 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 7c, has another effect: it correlates random choices in the generator that were previously independent. In particular, the generator in Figure 7a chooses between different constructors, namely `Leaf` and `Branch`, independently of the color of the parent node. But that is no longer the case in Figure 7c, which parametrizes symbolic weights by the `parentColor`. Depending on the `color` of the parent node, the probability of choosing `Leaf` changes.

[7]Using the notation from Section 3, a generator G with n symbolic parameters exhibits the space of distributions represented by $\{p_{G,w} \mid w \in [0,1]^n\}$.
[8]This is the classic problem of *underfitting*, which is well-studied in the machine learning literature [36].

```
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. 7. 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.

Algorithm 1: Gradient Descent to Maximize Objective.

Input: G: a generator with symbolic weights θ
f: a function that symbolically computes an objective function, given a generator distribution with symbolic weights
w_{initial}: initial value of symbolic weights
α, I: learning rate, number of iterations

Output: $\mathbf{w}_{\text{final}}$: tuned value of symbolic weights

1 $\mathbf{w} \leftarrow \mathbf{w}_{\text{initial}}$
2 **for** *iter* $= 1$ **to** I **do**
3     $\delta \leftarrow \nabla_\theta f(\llbracket G \rrbracket_L)\Big|_{\theta=w}$
4     $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta$
5 **end**
6 **return w**

The resulting dependency between random choices allows the generator to more closely fit an objective function. One can also add even more dependencies by *frontloading* random choices, allowing them to be made in tandem. For example, the RBT generator in Figure 7a makes independent choices for the constructors of the two children. One can frontload these choices and make them correlated as shown in Figure 7d. The `genTree` function below chooses the constructors for the two children beforehand and pass that information as arguments to the subcalls.

For our evaluation in Section 6 we use the techniques in from this section to increase the tunability of the generators. We describe our evaluation methodology in more detail there.

5 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 optimization algorithm, gradient descent, as described in Algorithm 1. Gradient descent requires computing gradients of the objective function (Line 3 of Algorithm 1) to determine the direction[9] in which to update the weights (Line 4).

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 mention 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.

5.1 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

[9]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*

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.

5.1.1 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 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 the $pr(\texttt{root})$ which gets further used in the computation of the objective function. Now, $pr(\texttt{root})$ can be computed recursively in a single bottom-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 [14, 17].

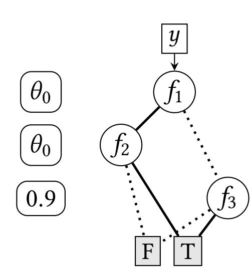
$$pr(\mathsf{T}) = 1, \qquad pr(\mathsf{F}) = 0, \qquad \text{(Base cases)}$$
$$pr(n) = w \cdot pr(h) + (1 - w) \cdot pr(l) \qquad \text{(Inductive Case)}$$

Example 5. *Consider the Loaded Dice program in Figure 8a. Loaded Dice compiles this program to the OBDD in Figure 8b. 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, $pr(root)$ can be computed as*

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


```
let x = flip1 θ0 in
let y = if x then flip2 θ0
        else flip3 0.9 in
y
```

(a) A Loaded Dice program with symbolic weights.



(b) Compiled BDD representing the program 8a.

Fig. 8. Compiling a Loaded Dice program to the OBDD upon which probabilistic inference and gradient computation is performed.

Note that the bottom-up traversal of an OBDD runs in time linear in the size of the OBDD [12]. 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.

5.1.2 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 computed in Example 5 are differentiable with respect to the symbolic weights. This is actually the case for exact probabilistic inference in Dice and Loaded Dice in general. As a result, probabilistic inference in Dice can support generator tuning via gradient descent.

The question that remains is how one can actually compute these gradients efficiently. The standard method in machine learning libraries [1, 11, 39] 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.

5.2 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 [36], as expectations can be approximated by averaging the expectand[10] over N samples. The resulting computation graph looks like Figure 9. 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.

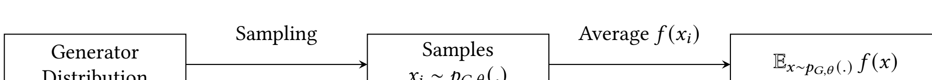


Fig. 9. The computation graph for approximating an expectation. To approximate $\mathbb{E}[f(x)]$, we can sample from the generator distribution and average $f(x)$ over the samples.

To resolve this issue, we adapt a well-known gradient estimation technique from the literature, REINFORCE [46], to our context of automated generator tuning. Specifically, for entropy based

[10] In the expression $\mathbb{E}_{x\sim p(.)}[f(x)]$, $f(x)$ is the expectand.

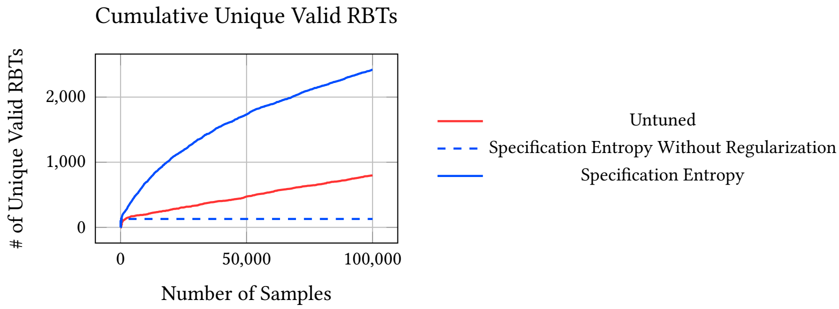


Fig. 10. Cumulative unique valid red-black trees throughout sampling, for our type-based RBT generator tuned for specification entropy, with and without regularization via bounded weights.

objective functions, we can estimate its gradient as follows. We include the derivation in Appendix B.

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

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 9. Note that in the computation of the expectand for each sample, Loaded Dice is used to compute an exact gradient at that sample.

5.3 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 within 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 10 that tuning with bounds allow the generators to achieve a much higher number of unique valid red-black trees as we obtain more samples.

6 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 3, Figure 4, and Figure 10 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 [7] and used it to test various case studies from the Etna benchmark suite [42]. Etna was specifically developed to evaluate PBT tools on their ability and speed to find bugs (pre-placed by the benchmark authors) in example programs.

6.1 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 (with additional dependencies as described in Section 4.2) the following strategies for these three workloads:

- **STLC, BST, and RBT Type-Based Generators**: We took the type-based generators that were derived automatically from type definitions using QuickChick and wrote them in Loaded Dice with additional parameters (as described in Section 4.2). We tuned these generators for diversity and validity via specification entropy using our approach. We compare these tuned generators against the untuned[11] 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 fix initial size parameters and introduce parameterize weights with the size parameters (Section 4.2). The generator is shown in Appendix A. We tune 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 tune 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.

6.2 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 11 and Figure 12. 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 Table 1. This training time is a one-time cost amortized over repeated testing. Once the generators are tuned, they can be used repeatedly for testing many programs.

6.3 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 11 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

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

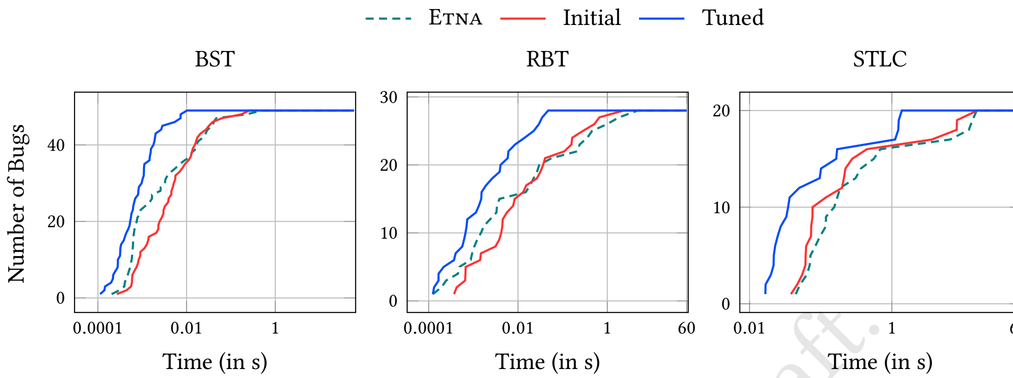
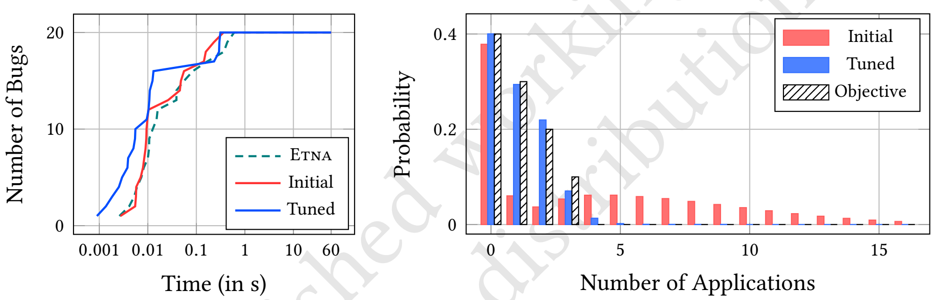


Fig. 11. Time in seconds vs. number of bugs found (higher is better) across workloads and generator strategies. The x-axes use log scaling.



(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.

Fig. 12. The results from tuning a bespoke STLC generator to leverage insight from Shi et al. [42]. The distribution of the number of applications is truncated for space (it goes up to 31, but all omitted values have a probability less than 0.005).

version in Etna (Table 1, Figure 12a). Additionally, Figure 12b validates that the distribution of applications did change as intended by tuning.

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.

7 Discussion

The previous section demonstrates how automated generator tuning can be useful in incorporating users' insights about the generator distribution and further helps in faster bug finding. In this section, we provide discussion and the larger context in which our approach fits.

7.1 Flat and Statically-Bounded Generators

Probabilistic programming languages such as Dice (and subsequently Loaded Dice) restrict the language they support in order to make their primary task of probabilistic inference tractable. In contrast, typical generator languages such as QuickCheck are much richer. As a result, not all generators are expressible in Loaded Dice. Loaded Dice, being a first-order language with only statically-bounded recursion, can only support 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 [35] and generic-random [32] fit this criteria, as do all specification-based generators derived from Rocq inductive relations in QuickChick [28, 37]. 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 $F_{<:}$.

7.2 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 as needed. 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 6, 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 11, and Figure 12). 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.

8 Related Work

We report on relevant related work for both generator tuning in property-based testing and probabilistic programming languages.

8.1 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 [16] and QuickChick [38] generator languages. This approach is simple, but it has the limitations we

Table 1. Speedup and training time of tuned generators over the original generators in Etna and the untuned generators with additional parameters.

Generator & Workload	Speedup vs. Etna	Speedup vs. Untuned	Training Time
BST Type-Based	3.5×	5.4×	3m
RBT Type-Based	5.8×	7.4×	3m
STLC Type-Based	4.3×	3.1×	7m
STLC Bespoke	2.3×	1.9×	8m

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 also auxiliary tools aimed at making the tuning process easier. In particular, Tyche [21] is a visual user interface that provides insights into the current distribution of a generator. This approach is complementary to Loaded Dice, as it could be used to confirm that a given objective had the intended effect or help to choose among multiple objectives. Automatic tuning could also enhance interfaces such as Tyche, such as by allowing the user to click-and-drag to adjust distributions.

Online tuning. One way to get around manual tuning is to tune during the testing process. The Target system [33], for example, uses algorithms like hill-climbing and simulated annealing during generation to maximize an objective function. In a similar vein, RLCheck [40] tunes generators with reinforcement learning, seeking out more diverse and valid inputs. The Choice Gradient Sampling algorithm [20] tunes online by manipulating the generator representation itself. ISLa [43] 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. [19] 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 techniques automatically derive well-distributed generators by construction, based on data types or inductive specifications. For example, DRaGeN [35] computes generator weights based on insights about branching processes, aiming to match a particular distribution of data constructors. This approach is great when the user does not already have a generator to start with, but it does not apply to tuning an existing generator; in those cases Loaded Dice is preferable. Additionally, Loaded Dice can work in concert with tools like DRaGeN, since the generators they produce may need to be further tuned: the user may decide a different distribution would be better, or they might want to specialize the distribution to a particular property. In these cases, Loaded Dice tuning can be applied to a synthesized generator to obtain a new set of weights for a new objective function.

8.2 Differentiating Discrete PPLs

A number of existing approaches bring together automatic differentiation and probabilistic programming. At a high level, the key novelty of Loaded Dice is its support for exact, reverse-mode automatic differentation for discrete probabilistic programs.

Gradient-Based Inference Algorithms. Probabilistic inference for arbitrary probabilistic programs is hard. This has led to the development of inference algorithms informed by gradients of specific quantities. Notably, Hamiltonian Monte Carlo [6] utilizes the gradient of the log posterior distribution, while variational inference [8, 9] utilizes automatic differentiation to estimate the posterior distributions of random variables in a probabilistic program. Both these approaches are limited to continuous probabilistic programs.

Gradient Estimators for Discrete Probabilistic Programs. Recent work such as ADEV [30] and `StochasticAD.jl` [4] propose program transformations to produce gradient estimators for probabilistic programs. These works offer different variance and scaling tradeoffs: while Loaded Dice uses OBDD-based exact computation of gradients, they employ Monte Carlo sampling to approximate these gradients. The objectives discussed in Section 3 and gradient estimator for entropy-based objectives (Section 5.2) are compatible with these systems.

Learning in Probabilistic Logic Programming. Loaded Dice compiles programs to binary decision diagrams and differentiates through them to learn weights. Similar functionality can be found in the realm of probabilistic logic programming. Languages like DeepProblog and Scallop [22, 31, 34] allow users to provide first-order logical specifications with weights that can be learned as outputs of neural networks. Loaded Dice is different from these approaches as it supports more traditional programming constructs, and we also provide specific learning objectives and associated algorithms to improve PBT.

9 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 the distribution of their generators 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. We empirically demonstrated that our technique enables 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.

10 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 Section 2, 3, 5 and 6. It will be submitted for Artifact Evaluation and will be made publicly available.

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

[7] 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

[8] 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

[9] 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

[10] Avrim Blum, John Hopcroft, and Ravindran Kannan. 2020. *Foundations of Data Science.* Cambridge University Press.

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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.

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

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

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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

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

[37] 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

[38] 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

[39] 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

[40] 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

[41] Claude Elwood Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423. http://plan9.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

[42] 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

[43] 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

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

[45] Wikipedia. 2024. Red–Black Tree — Wikipedia, The Free Encyclopedia.

[46] 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

[47] 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

A STLC Bespoke Generator

We include the "bespoke" generator for simply-typed lambda calculus terms tuned in Figure 2 and Section 6. It corresponds to the QuickChick generator from Etna [42], which is approximately 60 lines of Rocq, translated to Loaded Dice and modified with added weights that depend on size and fixed initial sizes.

```
@type Typ = TBool() | TFun(Typ, Typ)
@type Expr = Var(Nat) | Bool(Boolean) | App(Expr, Expr) | Abs(Typ, Expr)

genVar′(ctx, t, p, r) =
  @match ctx (
    Nil() → r,
    Cons(t′, ctx′) →
      @dice if t == t′
        genVar′(ctx′, t, p + 1, Cons(p, r))
      else
        genVar′(ctx′, t, p + 1, r)
      end
  )

genZero(env, tau) =
  @match tau (
    TBool() → Some(Bool(genBoolean())),
    TFun(t1,t2) →
      bindOpt(
        genZero(Cons(t1, env), t2),
        e -> Some(Abs(t1, e))
      )
  )

genTyp(s) =
  @match s (
    0 → TBool(),
    S(s′) → (
      w_bool, w_fun =
        @match s (
          1 → (θ_bool1, θ_bool1),
          2 → (θ_bool2, θ_fun2)
        );
      freq [
        w_bool ⇒ TBool(),
        w_fun ⇒ (
          t1 = genTyp(s′);
          t2 = genTyp(s′);
          TFun(t1, t2)
      ]
    )
  )

genExpr(env, tau, size) =
  @match size (
    0 → (
      backtrack [
        θ_0var ⇒ oneOf([
          None(),
          map(
```

```
          x → Some(Var(x)),
          genVar'(env, tau, 0, Nil())
        )
      ]),
      θ_0zero ⇒ genZero(env, tau)
    ],
  ),
  S(n) → (
    w_var, w_app, w_val = @match size (1 → (θ_var1,θ_app1,θ_val1), ..., 5 → (θ_var5,θ_app5,θ_val5));
    backtrack [
      (w_var,
        oneOf(
          None(),
          map(
            x → Some(Var(x)),
            genVar'(env, tau, 0, Nil())))),
      (w_app, (
        argty = genTyp(2);
        bindOption(genExpr(env, TFun(argty, tau), n),
          e1 →
            bindOption(genExpr(env,argty,n),
            e2 →
              Some(App(e1,e2)))))),
      (w_val,
        @match tau (
          TBool() → Some(Bool(genBool())),
          TFun(t1, t2) →
            bindOption(genExpr(cons(t1,env),t2,n),(e →
              Some(Abs(t1,e))))))
    ]
  )
)

G = genExpr(Nil(), genTyp(2), 5)
```

Appendix B follows on the next page.

B Adapting REINFORCE for Entropy Gradient Estimation

Proposition 1. *The gradient of the entropy of $p_{G,\theta}$, which can be expressed as the expectation of $\log p_{G,\theta}(x)$, can be estimated as follows:*

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

Proof.

$$\begin{aligned}
\nabla_\theta \mathop{\mathbb{E}}_{x\sim p_{G,\theta}(.)}[\log p_{G,\theta}(x)] &= \nabla_\theta \sum_{x\in X} p_{G,\theta}(x) \cdot \log p_{G,\theta}(x) \\
&= \sum_{x\in X} \nabla_\theta p_{G,\theta}(x) \cdot \log p_{G,\theta}(x) && \text{(Leibnitz Integral Rule)} \\
&= \sum_{x\in X} p_{G,\theta}(x) \cdot \nabla_\theta \log p_{G,\theta}(x) + \log p_{G,\theta}(x) \cdot \nabla_\theta p_{G,\theta}(x) \\
& && \text{(Product Rule of Differentiation)} \\
&= \sum_{x\in X} \nabla_\theta p_{G,\theta}(x) + \log p_{G,\theta}(x) \cdot p_{G,\theta}(x)\nabla_\theta \log p_{G,\theta}(x) \\
& && (\nabla_\theta \log p_{G,\theta}(x) = \tfrac{\nabla_\theta p_{G,\theta}(x)}{p_{G,\theta}(x)}) \\
&= \sum_{x\in X} \log p_{G,\theta}(x) \cdot p_{G,\theta}(x)\nabla_\theta \log p_{G,\theta}(x) \\
& && (\textstyle\sum_{x\in X} \nabla_\theta p_{G,\theta}(x) = \nabla_\theta \sum_{x\in X} p_{G,\theta}(x) = \nabla_\theta 1 = 0) \\
&= \mathop{\mathbb{E}}_{x\sim p_{G,\theta}(.)}[\log p_{G,\theta}(x)\nabla_\theta \log p_{G,\theta}(x)]
\end{aligned}$$

□