Automatic compile-time synthesis of entropy-optimal Boltzmann samplers

Maciej Bendkowski
maciej.bendkowski@gmail.com

Abstract

We present a famework for the automatic compilation of multi-parametric Boltzmann samplers for algebraic data types in Haskell. Our framework uses Template Haskell to synthesise efficient, entropy-optimal samplers generating random instances of user-declared algebraic data types. Users can control the outcome distribution through a pure, declarative interface. For instance, users can control the mean size and constructor frequencies of generated objects. We illustrate the effectiveness of our framework through a prototype `generic-boltzmann-brain` library showing that it is possible to control thousands of different parameters in systems of tens of thousands of ADTs. Our prototype framework synthesises Boltzmann samplers capable of rapidly generating random objects of sizes in the millions.

CCS Concepts: • **Theory of computation → Generating random combinatorial structures**.

Keywords: Boltzmann samplers, random generation

ACM Reference Format:
Maciej Bendkowski. 2022. Automatic compile-time synthesis of entropy-optimal Boltzmann samplers. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Consider the following example of a pair of algebraic data types **Lambda** and **DeBruijn** defining lambda terms in De-Bruijn notation [9]:

```
data DeBruijn
  = Z
  | S DeBruijn

data Lambda
  = Index DeBruijn
  | App Lambda Lambda
  | Abs Lambda
```

In the following paper we develop a general framework for compile-time generation of efficient Boltzmann samplers [12] for system of algebraic data types, such as **Lambda** and **DeBruijn**. Our prototype library[1] exposes a minimal, declarative Template Haskell [27] interface. For instance

```
mkDefBoltzmannSampler ''Lambda 10_000
```

declares **Lambda** an instance of the **BoltzmannSampler** type class:

```
class BoltzmannSampler a where
  sample :: RandomGen g =>
            Int -> MaybeT (BuffonMachine g) (a, Int)
```

The above type class defines types with a single `sample` function. Given an integer upper bound n, `sample` generates a random instance γ of type `a` together with its corresponding size $s \leq n$. While computing a random `a`, the generator consumes random bits provided within the **BuffonMachine** `g` monad. Because the generation process might sometimes fail, the whole computation is wrapped in a **MaybeT** monad transformer.

The `size` function satisfies two key Boltzmann sampler properties:

- instances of `a` with the *exact same size* have *the exact same* probability of being generated, and
- the *expected size* of the generated instances of `a` follows the user-declared value, such as 10,000 for **Lambda**.

In other words, while the size of the outcomes may vary, the outcome distribution is *fair*, *i.e. uniform* when conditioned on the size of the generated objects.

When a finer control over the outcome size is required, *rejection sampling* can be adopted *cf.* [3]:

```
rejectionSampler ::
  (RandomGen g, BoltzmannSampler a) =>
     LowerBound -> UpperBound -> BuffonMachine g a
```

Given two lower and upper bounds, a rejection sampler generates random instances of `a` until a sample of admissible size is generated. The expected runtime complexity of such a sampler depends on the *width* of the admissible size window. If it is an interval of the form $[(1-\varepsilon)n, (1+\varepsilon)n]$ for some positive *tolerance* parameter $\varepsilon > 0$, the runtime complexity of the rejection sampler is *linear*, *i.e.* $O(n)$. When the tolerance parameter ε is equal to 0, the rejection sampler returns

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] https://github.com/maciej-bendkowski/generic-boltzmann-brain

objects of some constant size n, and the expected runtime of `rejectionSampler` becomes $O(n^2)$, *cf.* [3, 12].

Compiled rejection samplers are readily available for use in property testing frameworks, such as QuickCheck [8]. For instance, `BuffonMachine` g computations can be easily converted to QuickCheck's `Gen` values:

```
quickCheckRejectionSampler ::
  BoltzmannSampler a =>
    (Int -> (LowerBound, UpperBound)) -> Gen a
```

By default, the *size* of generated objects is equal to the overall *weight* of constructors used in their construction. For instance, the size of `Abs (App (Index Z) (Index Z))` is equal to six as it consists of size constructor of default weight one. If such a *size notion* is not desired, it is possible to redefine the constructor weights, *e.g.* as follows:

```
mkBoltzmannSampler
  System
    { targetType = ''Lambda
    , meanSize = 10_000
    , frequencies = def
    , weights =
        ('Index, 0)
          <:> $(mkDefWeights ''Lambda)
    }
```

Note that here we declared a Boltzmann sampler for `Lambda` with (expected) mean size 10, 000, and a new set of *constructor weights* in which all constructors except `Index` have default weight one. The remaining `Index` constructor contributes now weight zero to the overall size of lambda terms.

1.1 Beyond uniform outcome distribution

In [4] a generalisation of Boltzmann samplers was introduced which lifted the classic univariate Boltzmann samplers to a *multi-parametric* setting. This multivariate paradigm is reflected in the presented framework in form of custom *constructor frequencies*. For instance

```
mkBoltzmannSampler
  System
    { targetType = ''Lambda
    , meanSize = 10_000
    , frequencies = ('Abs, 4_000) <:> def
    , weights =
        ('Index, 0)
          <:> $(mkDefWeights ''Lambda)
    }
```

declares a multi-parametric Boltzmann sampler for `Lambda` in which the target mean size is still 10, 000, however now we additionally require that the *mean weight contribution* of abstractions is equal to 4, 000.

The `size` function satisfies now the following generalised Boltzmann sampler properties:

- instances of `Lambda` with the exact same size *and* the same cummulative abstraction weights have *the exact same* probability of being generated, and
- the *expected size* of the generated instances is still 10, 000, whereas the *expected number* of abstractions is equal to the user-declared value of 4, 000.

It is therefore possible to *tune* the natural frequency of each constructor in `Lambda` and `DeBruijn` to one's needs. Note however that such an additional control causes a significant change in the underlying outcome distribution. In extreme cases, such as for instance requiring 80% of internal nodes in plane binary trees, the sampler might fail to compile or be virtually ineffective due to the sparsity of tuned structures.

1.2 Multiple Boltzmann sampler instances

Because Boltzmann samplers are implemented as instances of the `BoltzmannSampler` type class, we cannot have two distinct Boltzmann samplers for the same type `a`. In some circumstances, however, having multiple Boltzmann samplers with different constructor frequencies or even size notions might be beneficial. To enable such use cases, the presented framework lets users define Boltzmann samplers for `newtype`s of respective types.

For instance, in the following snippet we define a representation of so-called *binary lambda terms*, initially introduced by Tromp [29] for the purpose of using lambda calculus in algorithmic information theory (*cf.* also [18]):

```
newtype BinLambda = MkBinLambda Lambda

mkBoltzmannSampler
  System
    { targetType = ''BinLambda
    , meanSize = 12_000
    , frequencies = ('Abs, 3000) <:> def
    , weights =
        ('Index, 0)
          <:> ('App, 2)
          <:> ('Abs, 2)
          <:> $(mkDefWeights ''Lambda)
    }
```

The `BinLambda` type borrows the algebraic representation of `Lambda`. Custom weights for `App` and `Abs` reflect Tromp's recursive binary string representation of lambda terms:

```
  encode :: Lambda -> [Bool]
  encode = \case
    Abs t     -> False : False : encode t
    App lt rt -> False : True : encode lt ++ encode rt
    Index n   -> encode' n
  where
    encode' :: DeBruijn -> [Bool]
    encode' = \case
      S n' -> True : encode' n'
      Z    -> [True]
```

Note that the size of a binary lambda term corresponds to the length of the corresponding encoded binary string. In addition to a new size notion, `BinLambda` uses a different set of constructor frequencies, and mean size.

2 Univariate Boltzmann models

Let $\mathcal{S}$ be a set of objects endowed with an intrinsic *size* function $|\cdot|\colon \mathcal{S} \to \mathbb{N}$ with the property that that for all $n \in \mathbb{N}$, the set of objects of n in $\mathcal{S}$ is finite. For such a class of objects, the corresponding (univariate) *generating function* $S(z)$ is the power series $S(z)$ defined as

$$S(z) = \sum_{n\geq 0} s_n z^n \tag{1}$$

whose coefficients $(s_n)_{n\geq 0}$ denote the number of objects of size n in $\mathcal{S}$, *cf.* [30].

Given a real *control parameter* $x \in [0,1]$, a *Boltzmann model* [12] is a probability distribution in which the probability $\mathbb{P}_x(\omega)$ of generating an object $\omega \in \mathcal{S}$ satisfies

$$\mathbb{P}_x(\omega) = \frac{x^{|\omega|}}{S(x)}. \tag{2}$$

provided that $S(x)$ is finite[2].

Note that under such a model

- objects of equal size have equal probabilities, and
- the outcome size is varying random variable.

Indeed, note that the probability $\mathbb{P}_x(N = n)$ that the size N of a randomly generated object is equal to n satisfies

$$\mathbb{P}_x(N = n) = \frac{s_n x^n}{S(x)} \tag{3}$$

In other words, the outcome size distribution depends both on the control parameter x, as well as on the intrinsic size distribution in $\mathcal{S}$, see *e.g.* Figure 1.

In consequence, the control parameter x influences the *expected (mean) outcome size* $\mathbb{E}_x(N)$, as well as the standard deviation $\sigma_x(N)$:

$$\mathbb{E}_x(N) = x\frac{\frac{d}{dx}S(x)}{S(x)} \qquad \sigma_x(N) = \sqrt{x\frac{d}{dx}\mathbb{E}_x(N)} \tag{4}$$

Given access to the values of S and its derivative $\frac{d}{dx}S$ it is possible to use formula (4) and aptly choose a value of the control parameter x so to obtain a Boltzmann model with expect size of outcomes equal to a *target mean size* n. Even though, in general, explicit formulas or numerical oracles for $S(x)$ and $\frac{d}{dx}S(x)$ might not be readily available, we will soon see that for specifications corresponding to algebraic data types, we can construct efficient oracles and thus automatically find apt values for the control parameter.

2.1 Compiling Boltzmann samplers

Boltzmann samplers, realising the outcome Boltzmann model, follow closely the *sum-of-products* structure of ADTs.

[2]Generating functions corresponding to algebraic specifications discussed in the current paper are *analytic*, *i.e.* are convergent within some non-empty complex circle $|z| < \rho$ for $\rho \in \mathbb{R}$ depending only on the class $\mathcal{S}$.

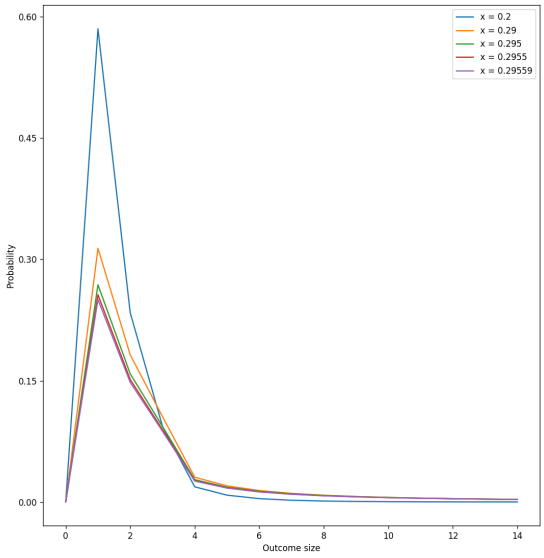

Figure 1. Example univariate Boltzmann models for `Lambda`. Note that the values $\mathbb{P}_x(N = n)$ quickly approach zero yet never reach it.

2.1.1 Singletons.

Consider a singleton class $\mathcal{S}$, *i.e.* a set consisting of a single element γ. Note that the corresponding generating function takes the form $S(z) = z^{|\gamma|}$. Consequently, the probability $\mathbb{P}_x(\gamma)$ of sampling γ is equal to one, and the respective Boltzmann sampler always returns γ.

2.1.2 Products.

Consider a product class $\mathcal{S}$ consisting of *pairs* $\gamma = (\alpha, \beta)$ where the components are arbitrary elements of classes $\mathcal{A}$ and $\mathcal{B}$. Under a Boltzmann model the probability $\mathbb{P}_x(\gamma)$ that γ is sampled satisfies

$$\mathbb{P}_x(\gamma) = \frac{x^{|\gamma|}}{S(x)} \tag{5}$$

Because the size of γ is equal to the total size of its components we can rewrite (5) as

$$\mathbb{P}_x(\gamma) = \frac{x^{|\gamma|}}{S(x)} = \frac{x^{|\alpha|+|\beta|}}{S(x)} = \frac{x^{|\alpha|}x^{|\beta|}}{S(x)} \tag{6}$$

Using Cauchy's product formula for power series

$$\left(\sum_{n\geq 0} a_n z^n\right)\cdot\left(\sum_{n\geq 0} b_n z^n\right) = \sum_{n\geq 0} c_n z^n \quad \text{where} \quad c_n = \sum_{k=0}^{n} a_k b_{n-k}$$

we notice that $S(z) = A(z)B(z)$ and therefore

$$\mathbb{P}_x(\gamma) = \frac{x^{|\alpha|}x^{|\beta|}}{S(x)} = \frac{x^{|\alpha|}x^{|\beta|}}{A(x)B(x)} = \mathbb{P}_x(\alpha)\mathbb{P}_x(\beta) \tag{7}$$

It means that in order to generate a random pair γ corresponding to $\mathcal{S}$ using a Boltzmann sampler, we can invoke

Boltzmann samplers for $\mathcal{A}$ and $\mathcal{B}$, and then return a pair of their results.

2.1.3 Coproducts. Consider a coproduct class $\mathcal{S}$ which is a *disjoint sum* of two classes $\mathcal{A}$ and $\mathcal{B}$. In other words, $\mathcal{S}$ consists of elements γ which belong to either $\mathcal{A}$ or $\mathcal{B}$ but not both at the same time. Note that in such a case the probability $\mathbb{P}_x(\gamma \in \mathcal{A})$ that an arbitrary object γ in $\mathcal{A}$ is sampled satisfies

$$\mathbb{P}_x(\gamma \in \mathcal{A}) = \frac{A(x)}{S(x)} \tag{8}$$

as

$$A(x) = \sum_{\gamma \in \mathcal{A}} x^{|\gamma|} \tag{9}$$

It means that in order to generate a random object γ in $\mathcal{S}$ using a Boltzmann sampler, we have to make a *skewed* coin toss. With probability $\frac{A(x)}{S(x)}$ we invoke the sampler corresponding to $\mathcal{A}$, and with probability $\frac{B(x)}{S(x)}$ we invoke the sampler corresponding to $\mathcal{B}$.

2.1.4 Algebraic data types. The above simple Boltzmann sampler compilation rules can be readily applied to concrete algebraic data types. Consider our running example system of two ADTs `Lambda` and `DeBruijn`. A Boltzmann sampler for `Lambda` has to make a random decision which constructor to use, *i.e.* `Abs`, `App`, or `Index`. If `Abs` is chosen, the `Lambda` Boltzmann sampler has to invoke itself, generate a random lambda term `lt`, and output `Abs lt`. Likewise, if `App` is chosen, the `Lambda` Boltzmann sampler has to invoke itself twice, generating two random lambda terms `lt` and `lt'`, and output `App lt lt'`. Finally, if `Index` is chosen, the `Lambda` Boltzmann sampler has to invoke the Boltzmann sampler for `DeBruijn` which will return a random `DeBruijn` index, and wrap it around `Index`. The Boltzmann sampler for `DeBruijn` is constructed similarly.

Let us remark that while Boltzmann samplers readily apply to algebraic data types, they are not limited to them. Over the years Boltzmann samplers have enjoyed a series of extensions and improvements including, *inter alia*, the support for so-called labelled [12], Pólya [14], or first-order differential specifications [5].

3 Multivariate Boltzmann models

The classical, univariate Boltzmann model controls a single system parameter, *i.e.* the expected outcome size. In some circumstances, however, a finer control over the outcome distribution is required. Multivariate Boltzmann models, initially introduced in [4], address this issue by generalising classical Boltzmann models to a multivariate setting in which multiple outcome parameters can be controlled simultaneously[3].

[3]Let us remark that, unless NP = RP, controlling the *exact* values of multiple parameters is practically infeasible, see [1].

Analogously to their univariate counterparts, multiparametric Boltzmann models depend on *multivariate generating functions*. A multivariate *generating function* $S(z_1, \ldots, z_d)$ is a power series $S(z_1, \ldots, z_d)$ defined as

$$S(z_1, \ldots, z_d) = \sum_{n_1, \ldots, n_d \geq 0} s_{n_1, \ldots, n_d} z_1^{n_1} \cdots z_d^{n_d} \tag{10}$$

whose coefficients $\left(s_{n_1, \ldots, n_d}\right)_{n \geq 0}$ denote the number of objects with n_i atoms of type z_i in $\mathcal{S}$, *cf.* [16]. For instance, z_1 can correspond to the *size* of lambda terms in `Lambda`, whereas z_2 can denote the number of its abstractions. Then, the coefficient $s_{n,k}$ denotes the number of lambda terms of size n which have k abstractions in total.

Given a vector of real control parameters $\vec{x} = (x_1, \ldots, x_d)$, a *multivariate Boltzmann model* is a probability distribution in which the probability $\mathbb{P}_{\vec{x}}(\omega)$ of generating an object $\omega \in \mathcal{S}$ with n_i atoms of type z_i satisfies

$$\mathbb{P}_{\vec{x}}(\omega) = \frac{x_1^{n_1} \cdots x_d^{n_d}}{S(x_1, \ldots, x_d)}. \tag{11}$$

The expected number $\mathbb{E}_{\vec{x}}(N_i)$ of atoms of type n_i satisfies

$$\mathbb{E}_{\vec{x}}(N_i) = x_i \frac{\frac{\partial}{\partial_{x_i}} S(\vec{x})}{S(\vec{x})} \tag{12}$$

Note that this is a straightforward generalisation of (4).

While compilation rules for univariate Boltzmann samplers readily generalise onto multiparametric samplers, *cf.* [1, 4], finding apt values for the d-dimensional *control vector* $\vec{x}$ poses an even more challenging problem.

4 Parameter tuning

The *key* to compiling Boltzmann samplers with expected outcome parameters lies in finding the value of the corresponding control vector $\vec{x}$ and the values of respective generating functions at $\vec{x}$. We call this process *parameter tuning*.

In simple systems, such as in our single-parameter running example of `Lambda` and `DeBruijn`, we have access to analytic closed form expressions for all the generating functions. Using the so-called *symbolic method* [16] we can lift the algebraic type definitions onto the level generating functions corresponding to the intrinsic size of objects in the associated classes. Unfortunately, for most systems of algebraic data types we do not have access to closed form expressions of respective generating functions. Therefore, in general, we have to resort to *numerical oracles*, instead.

For systems without additional tuning parameters we could use a quickly convergent Newton iteration procedure developed in [25]. For generalised systems with d tuning parameters, on the other hand, we could use a generalised Newton iteration scheme developed in [4]. Unfortunately, the latter is impractical both due to its exponential $O(n^{1+\frac{d}{2}})$ running time, as well as the fact that the iteration is convergent in an *a priori* unknown d-dimensional vicinity of the target control vector $\vec{x}$ value.

In the actual implementation of the presented framework we resort to an alternative method based on convex optimisation techniques.

4.1 Convex optimisation

We illustrate the principle of *tuning as convex optimisation* [1] on our running example of `Lambda` and `DeBruijn` where we request a Boltzmann model for lambda terms with mean size 10,000 and 2,500 abstractions in expectation. We assume a size notion in which the constructor `Index` contributes weight zero and all other constructors contribute weight one. Let us recall the system under consideration:

```
data DeBruijn
  = Z
  | S DeBruijn

data Lambda
  = Index DeBruijn
  | App Lambda Lambda
  | Abs Lambda
```

Let us denote the generating function corresponding to `Lambda` and `DeBruijn` by $L(z)$ and $D(z)$, respectively. Using (12) we can formulate an *optimisation problem* based on (15):

$$\text{Minimise} \qquad (13)$$

$$\left(z\frac{\frac{\partial}{\partial_z}L(z,u)}{L(z,u)} - 10,000\right) + \left(u\frac{\frac{\partial}{\partial_u}L(z,u)}{L(z,u)} - 2,500\right)$$

for z, u.

In other words, we ask for z, u which result in a Boltzmann model in which the expected size of lambda terms is 10,000 and the mean number of abstractions is equal to 2,500.

Unfortunately, in such a form the optimisation problem (13) is too general to use an optimisation solver. Following [1] we therefore reformulate it as a *convex optimisation problem* exploiting the regular structure of algebraic data types `Lambda` and `DeBruijn`.

We start with mapping the system to a system of corresponding (univariate) generating functions using the symbolic method [16]:

$$\begin{aligned} D(z) &= z + zD(z) \\ L(z) &= D(z) + zL(z)^2 + zL(z) \end{aligned} \qquad (14)$$

The transformation is purely mechanical and follows the structure of involved algebraic type definitions.

We start with `DeBruijn`. It has two constructors which generate *distinct* inhabitants of `DeBruijn`. We can therefore think of `DeBruijn` as a disjoint sum of two classes of objects, *i.e.* the singleton class `Z`, and the class `S DeBruijn` of successors. The former class has a single inhabitant of size one, hence its generating function is just z. The latter class, on the other hand, consists of DeBruijn indices in the form of `S n` where `n` is itself a DeBruijn index. The topmost constructor `S` contributes weight one to each of the indexes, and so the corresponding generating function takes form $zD(z)$ where $D(z)$ is the generating function for DeBruijn indices.

Next, let us consider `Lambda`. Its type definition consists of three constructors which give rise to three distinct classes, *i.e.* indices, applications, and abstractions. Because `Index` contributes no weight, the respective generating function is $D(z)$. On the other hand, `App` and `Abs` contribute weight one, and so the corresponding generating functions for applications and abstractions take forms $zL(z)^2$ and $zL(z)$, respectively. Note that the exponent of $L(z)$ corresponds to the arity of the respective constructor. In general, each constructor definition `T` $a_1 \dots a_k$ can be thought of as a generalised product $(\cdots(\mathtt{T}\ a_1)\dots a_k)$. Consequently, the corresponding generating function is of form $z^w A_1(z)\cdots A_k(z)$ where w is the weight of `T`, and $A_1(z), \dots, A_k(z)$ are the generating functions corresponding to the respective argument types.

Next, for each custom constructor frequency we create a new *marking variable* and place it in the definition of the respective generating function:

$$\begin{aligned} D(z,u) &= z + zD(z,u) \\ L(z,u) &= D(z,u) + zL(z,u)^2 + zuL(z,u) \end{aligned} \qquad (15)$$

Note that u marks now occurrences of abstractions. In other words, the coefficient $l_{n,k}$ standing by $z^n u^k$ in the generating function $L(z,u)$ denotes the number of lambda terms of size n and k abstractions.

At this point, we have successfully mapped our example system of algebraic data types into a corresponding system of multivariate generating functions. Symbolically, our system of multivariate generating functions takes the general form $\vec{F} = \vec{\Phi}(\vec{F}, \vec{Z})$ where $\vec{F}$ denotes the vector of generating functions, $\vec{\Phi}$ denotes the vector of corresponding right-hand side expressions, and $\vec{Z}$ stands for the vector of (all) tuning variables.

First, for $\vec{F} = (L(z,u), D(z,u))$ and $\vec{Z} = (z,u)$ we introduce new variables, *i.e.* $\vec{f} = (\lambda, \delta)$ and $\vec{z} = (\zeta, \upsilon)$, respectively. Next, we apply the following *log-exp transformation* [1] to (15)

$$\vec{F} = \vec{\Phi}(\vec{F}, \vec{Z}) \longrightarrow \vec{f} \geq \log\left(\vec{\Phi}(\exp(\vec{f}), \exp(\vec{z}))\right) \qquad (16)$$

resulting in

$$\begin{aligned} \delta &\geq \log\left(e^{\zeta} + e^{\zeta+\delta}\right) \\ \lambda &\geq \log\left(e^{\delta} + e^{\zeta+2\lambda} + e^{\zeta+\upsilon+\lambda}\right) \end{aligned} \qquad (17)$$

The above two inequalities form *convex* optimisation constraints. What remains is to formulate the optimisation goal. In general the optimisation goal takes form

$$f_o - \vec{\mu}^{\intercal}\vec{z} \longrightarrow \min_{\vec{f},\vec{z}} \qquad (18)$$

where f_o is the *target type* whose inhabitants we indend to generate, and $\vec{u}$ is a vector of user-declared expectations

matching $\vec{z}$ and thus the introduced tuning parameters. In our running example the optimisation goal becomes

$$\lambda - 10,000\zeta - 2,500\upsilon \longrightarrow \min_{\lambda,\delta,\zeta,\upsilon} \tag{19}$$

In the end, we constructed a complex optimisation problem from whose solution we can recover the values for z, u and, $L(z), D(z)$ which realise the desired Boltzmann model, *cf.* [1].

4.2 Complexity

The *parameter tuning* process goes through a few phases, *i.e.* the problem formulation, running an convex optimisation solver, recovering the value of the control parameter and respective genrating functions, and finally computing the constructor probabilities for each constructor in the considered system. The single most expensive phase is finding a proper solution to the convex optimisation problem.

Luckily, due to the regular shape of algebraic data types, we can leverage polynomial interior-point algorithms for convex optimisation [23] and use practically feasible solvers to achieve parameter tuning. In our current framework, we rely on an external library called `paganini` [1] which allows us to model tuning as a *disciplined convex optimisation problem* (DCP) [17]. The DCP modelling framework can be viewed as a *domain specific language* which allows its users to systematically build convex optimisation problems out of simple expressions such as $\log \cdot$ and $\log \sum \exp \cdot$ though a set of composition rules which follow basic convex analysis principles. The framework takes care of most tedious tasks such as formulating the problem in standard form, or providing a feasible starting point to the solver. While DCP covers a strict subset of the interior-point framework using so-called conic solvers, problems stemming from algebraic data types can be effectively expressed and solved.

The authors of [1] report a benchmark example of a transfer matrix model tuned using `paganini`. It consists of a Boltzmann model generating $n \times 9$ polyomino tillings with over 1,000 different available tiles. Each tile has a distinct colour and shape, see Figure 2. The model was tuned so to achieve

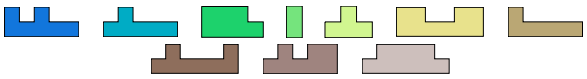

Figure 2. Examples of admissible tiles.

outcome polyomino tillings with a *uniform* colour palette, *i.e.* each colour occupies in expectation the same amount of space in each tilling, *cf.* Figure 3. Polyomino tillings of this form have a corresponding *finite state automaton* with 19,000 states and 357,000 transitions which roughly corresponds to a system of 19,000 (mutually recursive) algebraic data types with a total of 357,000 different constructors.

Let us remark that we need to approximate the control vector $\hat{x}$ with precision of order $O(\frac{1}{n})$ to obtain a rejection

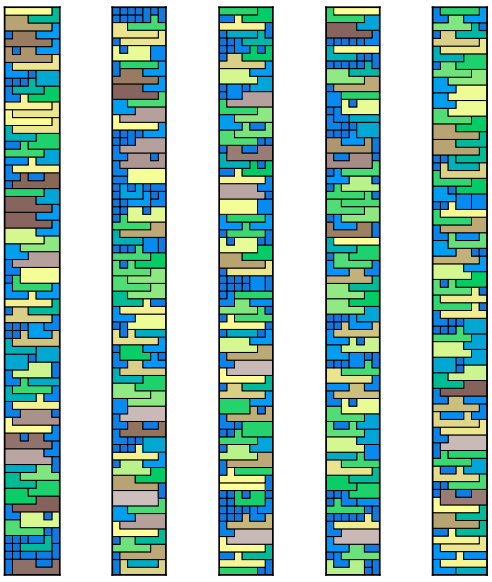

Figure 3. Five random $n \times 7$ tillings of areas in the interval $[500; 520]$ using in total 95 different tiles.

sampler with linear time complexity. For a more detailed analysis we invite the curious reader to [1].

5 Architecture overview

Given a system such as

```
mkBoltzmannSampler
  System
    { targetType = ''Lambda
    , meanSize = 10_000
    , frequencies = ('Abs, 4_000) <:> def
    , weights =
        ('Index, 0)
          <:> $(mkDefWeights ''Lambda)
    }
```

the presented framework takes the following steps:

1. compute the control vector parameters and related generating function values for the user-declared parameter values, and
2. compile a Boltzmann sampler realising the computed Boltzmann model.

5.1 Computing constructor distributions

We convert the system declaration into a `paganini` input using a custom monadic EDSL we call `paganini-hs`[4]:

```
formProblem ::
  IO (Either PaganiniError [Maybe Double])
formProblem = paganini' @@ do
  Let z <- variable' 10_000
  Let u <- variable' 4_000
```

[4] https://github.com/maciej-bendkowski/paganini-hs

```
  Let d <- variable
  Let l <- variable

  -- D(z,u) = z + zD(z,u)
  d .=. z + z * d

  -- L(z,u) = D(z,u) + zL(z,u)^2 + zuL(z,u)
  l .=. d + z * l^2 + z * u * l

  tuneAlgebraic l -- tune for l
  value <$> [z, u, l, d]
```

Note that the monadic computations inside `formProblem` follow the transformation outlined in (15).

We start with introducing two marking variables `z` and `u` which correspond to the size of generated lambda terms and the number of their abstractions, respectively. These variables are initialised with user-declared values. Next, we introduce two more variables `d` and `l` which correspond to **`Lambda`** and **`DeBruijn`**, respectively. Note that these variables are not *tuned*.

Next, we proceed with mapping type definitions to their corresponding generating function definitions. Variable `d` denotes $D(z,u)$ whereas `l` denotes $L(z,u)$. After formulating the definitions of the type variables in the input system we invoke `paganini` using `tuneAlgebraic l`, and output the tuned values of $z, u, L(z,u), D(z,u)$.

The whole computation is wrapped in a specification monad **`Spec`** which formulates a corresponding problem in the input format of `paganini`[5]:

```
import paganini as pg
spec = pg.Specification()
z = pg.Variable(10000)
d = pg.Variable(4000)
l = pg.Variable()
d = pg.Seq(z)

spec.add(l, d + z * u * l + z * l**2)
spec.run_tuner(l, method = pg.Method.STRICT)

print (z.value)
print (u.value)
print (l.value)
print (d.value)
```

`Paganini` itself is a `python` DSL written on top of `CVXPY` [11] — a modelling library for disciplined convex optimisation. It lets us express the tuning problem in a convenient, domain specific form which can be then transformed into a readily solvable convex optimisation problem. Note that in doing so, our framework does not need to formulate the problem directly, but rather can treat `paganini` as a black-box solver.

[5]For presentation purposes we elide boilerplate code handling, e.g. error handling. The actual input is slightly more involved.

Let us mention that we use `BinAnn` [21] to easily transfer variables names from `paganini-hs` to `paganini`. All variable names are reflected in both DSLs. Such a design decision makes debugging easier and lets `paganini-hs` provide better error messages with meaningful variable names.

5.2 Sampling from discrete distributions

Once the values of the control vector and corresponding generating functions are computed, we can readily calculate the *branching probabilities* for involved types. Given a type `a`, the respective Boltzmann sampler for `a` has to make a random decision determining which constructor to use in the process of generating a random object in `a`. In our running example, the *branching probabilities* for type **`Lambda`** take the form

$$\frac{D(z,u)}{L(z,u)} \quad \frac{zuL(z,u)}{L(z,u)} \quad \frac{zL(z,u)^2}{L(z,u)} \tag{20}$$

for **`Index`**, **`Abs`**, and **`Abs`**, respectively, and aptly chosen values of z and u. Hence, in order to choose a constructor we have to draw a random variable from a *discrete probability distribution*. To do so, we can resort to the well-known *inversion method* [10]; we partition the interval $[0,1]$ into three segments, each of length corresponding to one of the available constructors, draw a random real p in between zero and one, and determine in which segment does p fall into.

While it is possible to choose random constructors using the inversion scheme, let us remark that it is quite inefficient for our purposes. The inversion method works well under the (unrealistic) real RAM model in which we operate on real numbers. In practice, we do not have *arbitrary* precision real numbers, but rather *finite* precision floating-point numbers. The inversion scheme samples therefore a random double-precision floating point number $p \in [0,1]$ to select one of the distribution points. In some cases the available precision of a single floating-point number might be not enough. In others, fewer bits are sufficient. For instance, note that in order to sample from a distribution $\left(\frac{1}{2}, \frac{1}{2}\right)$ a *single* bit is sufficient.

Due to these limitations, we do not use the inversion scheme but rather resort to a different approach following the *random bit model of sampling* introduced by Knuth and Yao in [19]. Instead of using a single floating-point number to sample from a discrete distribution, one accesses a *lazy* stream of random bits, consuming one bit at a time. These bits are then used to refine the search space until a single value can be chosen.

For performance reasons, in our implementation we do not use an actual stream of bits, but rather use a buffered oracle [20].

```
data Oracle g = Oracle
  { buffer :: !Word32
  , usedBits :: !Int
  , rng :: g
  }
```

The oracle type `Oracle g` is parameterised by a random number generator `g`. The oracle consists of a 32-bit buffer and a counter keeping track of how many random bits have been consumed so far from the current buffer. If the buffer gets depleted, it can be regenerated as follows

```
fresh :: RandomGen g => g -> Oracle g
fresh g = case random g of
(x, g') -> Oracle { buffer = x
                  , usedBits = 0
                  , rng = g'
                  }
```

Using the `Oracle` type, we can now define a `BuffonMachine`[6] monad for random computations in the random bit model framework. The `BuffonMachine` type is implemented as a `newtype` wrapper around the `State` monad:

```
newtype BuffonMachine g a = MkBuffonMachine
  {runBuffonMachine :: State (Oracle g) a}
  deriving (Functor, Applicative, Monad)
    via State (Oracle g)
```

Using the ideas of [19] it is possible to construct an entropy-optimal *discrete distribution-generating tree* (DDG) implementing a sampler for any discrete distribution P of rational numbers. In other words, it is possible to construct a sampler for P which uses the least average number of random bits to sample from P. Unfortunately, the entropy-optimal DDGs can be *exponentially* large in the number of bits required to encode the input distribution P. For instance, the binomial distribution $\mathbb{B}(n, p)$ with parameters $n = 50$ and $p = \frac{61}{500}$ requires a DDG of height 10^{104}, see [26].

Unfortunately, such an overhead renders DDGs impractical for our purposes. Due to that, we use recently developed *approximate sampling schemes* [26] which are a practical trade-off between the entropy perfect DDGs and feasible, finite precision sampling algorithms. Instead of sampling from a discrete probability distribution $P = (p_1, \ldots, p_n)$ we find an entropy optimal sampling algorithm for a closest approximation $\hat{P} = (\hat{p}_1, \ldots, \hat{p}_n)$ of P among all sampling algorithms which operate within a finite k-bit precision. Let us note that the framework of approximate sampling schemes, and in particular its prototype implementation[7], supports several statistical measures of *approximation error* between probability distributions, including Kullback-Leibler, Pearson chi-square, and Hellinger divergence.

The optimal approximate distribution $\hat{P}$ can be readily found as soon as the constructor distribution is computed in so-called linear, compact vector form. We use prototype implementation of optimal approximate sampling algorithms to find the compact vector form of DDGs. Compiled Boltzmann samplers readily choose constructors from the compact DDGs represented as `Vector Int`.

5.3 Anticipated rejection

A straightforward implementation of Boltzmann samplers

```
sample :: RandomGen g => BuffonMachine g a
```

has some practical drawbacks. While the underlying Boltzmann model provides control over the *mean size* of its outcomes, we have no finer control over the *actual* size of generated objects. In some cases, the outcome sample size might be significantly larger than the user-declared mean size. Without any additional control, Boltzmann samplers might consume significantly more resources than required.

In the presented framework we implement Boltzmann samplers with *anticipated rejection*, see [2]. The idea is quite simple. The user provides an *upper bound* on the size of generated outcomes[8]. During generation we maintain the current size of the sample. If it exceeds the given upper bound, the process is terminated and the sample is *rejected*. Consequently, the signature of `sample` becomes

```
sample :: RandomGen g =>
          Int -> MaybeT (BuffonMachine g) (a, Int)
```

To give the user a more fine-grained control over the outcome size of sampled objects, the user can provide an *admissible* size range $[(1-\varepsilon)n, (1+\varepsilon)]$. The framework samples objects until one with admissible size is generated. Note that such a *rejection* scheme guarantees that inadmissible samples are rejected as soon as possible.[9]

```
toleranceRejectionSampler ::
  (RandomGen g, BoltzmannSampler a) =>
  Int -> Double -> BuffonMachine g a
toleranceRejectionSampler n eps = rejectionSampler lb ub
  where
    lb = MkLowerBound $
      floor $ (1 - eps) * fromIntegral n
    ub = MkUpperBound $
      ceiling $ (1 + eps) * fromIntegral n
```

Let us remark that anticipated rejection has an impact on the underlying Boltzmann model. Note that as we limit admissible sizes, we impose a small bias in the distribution, initially not taken into account. Consequently, we have to modify the original tuning goal $\mathbb{E}(N) = n$ to accommodate an additional *bias parameter* δ such that $\mathbb{E}(N) = \delta n$, *cf.* [1]. The specific value of δ depends on the type of parameter corresponding to the variable N and, in particular, its corresponding asymptotic behavior in the related system of multivariate generating functions. Introducing the bias factor can diminish the number of rejections required to find

[6] The name Buffon machine was coined by Flajolet, Pelletier and Soria who studied probability distributions which can be simulated *perfectly* using a source of unbiased random bits [15]. While we do not make direct use of their ideas, we consider them a source of inspiration for our current work.

[7] https://github.com/probcomp/optimal-approximate-sampling

[8] Note that this is also the recommended generator design choice of QuickCheck.

[9] The same idea can be readily applied to *all* parameters. Our prototype framework supports only anticipated rejection for the outcome size.

an admissible sample. For more details we invite the curious reader to [1].

5.4 Newtype samplers

Boltzmann samplers for algebraic data structures have a regular format. For instance, our running example of `Lambda` has the following Boltzmann sampler:

```
instance BoltzmannSampler Lambda where
  sample ub =
    do guard (ub >= 0)
      lift (BuffonMachine.choice ...)
          >>=
            (\case
              0 -> do (x_0, w_0) <- sample ub
                      pure (Index x_0, w_0)
              1 -> do (x_0, w_0) <- sample (ub - 1)
                      (x_1, w_0) <- sample (ub - w_0 - 1)
                      pure (App x_0 x_1, 1 + w_0 + w_1)
              2 -> do (x_0, w_0) <- sample (ub - 1)
                      pure (Abs x_0, 1 + w_0)
```

The `sample` function has a single parameter `ub` which defines a *size budget* which the sampler cannot overreach, as guarded by the `guard (ub >= 0)` expression. If the sampler has some positive size budget, it can proceed with generating the object. To do so, the sampler draws a random number according to the respective constructor distribution. The choice function has signature

```
choice :: RandomGen g => Distribution -> Discrete g
```

where

```
newtype Distribution =
  MkDistribution {unDistribution :: Vector Int}
  deriving stock (Show)

type Discrete g = BuffonMachine g Int
```

represent the compact linear DDG, and discrete random integer variables. Note that the actual distribution is inserted directly in the body of the sampler function.

Next, the generated random number is mapped onto a concrete constructor. We use `sample` to generate all of the constructor parameters. At the same time, we keep track of the size budget accounting for the weight of the considered constructor and size of each generated subexpression.

Such a Boltzmann sampler construction easily generalises onto arbitrary algebraic data types. However, since samplers are implemented as instances of the `BoltzmannSampler` type class, we can have at most one sampler for each type. In some circumstances, we might want to have multiple samplers for the same type. To support such use cases, we support the compilation of Boltzmann samplers for `newtype` synonyms. Note that the structure of such Boltzmann samplers is *almost* the same as for regular data types. To support them, we need to change the constructor distribution, and adjust the return type of generated object. The former can be achieved through a separate tuning problem. The latter, on the other hand, through safe, zero-cost constructor type coercions [6]. For each constructor application we introduce an explicit coercion which changes the constructor type so match the `newtype` synonym. For instance, for λ-term application we use

```
coerce
  @(Lambda -> Lambda -> Lambda)
  @(BinLambda -> BinLambda -> BinLambda) App
```

instead of `App`. Note that such a coercion imposes correct type constraints on the argument `sample` corresponding to the considered constructor.

5.5 Known limitations

Multi-parametric Boltzmann samplers support systems of (possibly mutually recursive) *non-parametric* algebraic data types, *i.e.* ADTs of kind $*$. Parametric ADTs, such as

```
data BinTree a
  = Node (BinTree a) (BinTree a)
  | Leaf a
```

of kind `(* -> *)` do not have a corresponding Boltzmann model as, *a priori*, the structure and size of objects of type `a` are unknown. Depending on the concrete instantiation of `a`, the constructor distribution for `BinTree` can take different forms. While it is possible to define Boltzmann models for `BinTree a` for non-parametric types `a`, our current prototype implementation does not support them.

Moreover, our framework does not provide default Boltzmann samplers for certain primitive types, such as `Bool` or `Integer`. The former is a type with finitely many inhabitants and thus Boltzmann models should not be preferred[10]. While the latter is an *infinite type*, there is no universal or default size notion attached to integers. In certain contexts a unary encoding of integers might be used, as for instance in the case of λ-terms in the DeBruijn notation, whereas in others a compact binary one might be more appropriate. Consequently, we do not impose a default size notion and leave the decision to the user.

Let us also remark that for certain size notions or requested constructor frequencies there might be no corresponding Boltzmann model. For instance, consider

```
data BinTree
  = Node BinTree BinTree
  | Leaf
```

where `Node` contributes weight one and `Leaf` contributes no weight at all. In other words, the number of `BinTree`s of size n corresponds to the nth Catalan number. While `BinTree` has a well-defined Boltzmann model under the assumed size notion, `[BinTree]` does not. Note that there is an infinite number of lists of `BinTree`s of size zero —

`[Leaf]`, `[Leaf, Leaf]`, `[Leaf, Leaf, Leaf]`, ...

[10] Let us notice that it *is* possible to define Boltzmann models for finite types, but other, more direct and exact sampling methods are available.

There exists therefore no *uniform* distribution of `BinTree` lists of size n and so, there is no corresponding Boltzmann model for `[BinTree]`. Such systems are called *ill-founded* and can, in principle, be recognised before the tuning procedure is initiated, *cf.* [25].

6 Benchmarks

To benchmark the run-time performance of our prototype implementation we use the following example system of λ-terms in DeBruijn notation:

```
data DeBruijn
  = Z
  | S DeBruijn

data Lambda
  = Index DeBruijn
  | App Lambda Lambda
  | Abs Lambda

mkBoltzmannSampler
  System
    { targetType = ''Lambda
    , meanSize = 1000
    , frequencies = def
    , weights =
        ('Index, 0)
          <:> $(mkDefWeights ''Lambda)
    }

lambdaSampler :: Int -> IO [Lambda]
lambdaSampler n =
  evalIO $
    replicateM n $
      rejectionSampler @SMGen
        (MkLowerBound 800) (MkUpperBound 1200)
```

We request a (univariate) Boltzmann sampler tuned so to generate random λ-terms with expected size 1,000. We measure the performance of a rejection sampler generating λ-terms of sizes in between 800 and 1,200. In other words, we tolarate a 20% size deviation from the expected target size.

We present three sets of criterion[11] benchmarking suites, generating 10, 100, and 1,000 random samples:

mean time	10.95 ms
standard deviation	882.9 μs

mean time	104.8 ms
standard deviation	5.67 ms

mean time	1.127 s
standard deviation	40.73 ms

Note that generating a single λ-term of size in between 800 and 1,200 takes, on average, around 1.1 ms.

An analogous sampler generating 100 samples of target mean size 10,000 and a smaller 10% tolerance has a similar performance:

mean time	3.064 s
standard deviation	119.5 ms

In this case, generating a single λ-term of size in between 9,000 and 11,000 takes, on average, 30 ms.

Generating 100 terms of even larger mean size 100,000 and the same, 10% tolerance gives the following benchmark:

mean time	26.16 s
standard deviation	1.371 s

Note that the sampler performance scales linearly with the target mean size. Generating a single λ-term of size in between 90,000 and 110,000 takes, on average, 260 ms.

Finally, we present a benchmark example generating 10 random λ-terms of mean size 1,000,000 and a 10% size tolerance:

mean time	42.07 s
standard deviation	8.128 s

Note that, on average, sampling a random λ-term takes just 4.2 s. It is therefore feasible to generate even larger λ-terms.

7 Related work

Boltzmann samplers. Automatic compilation of Boltzmann samplers for algebraic data types was first implemented in Objective Caml [7]. While similar in spirit to the presented work, compiled samplers do not support multi-parametric tuning. Branching probabilities are computed using a combinatorial Newton method developed in [25].

A similar `boltzmann-samplers` framework for the automatic compilation of Boltzmann samplers was developed for Haskell[12]. This framework, however, does not support multi-parametric tuning. It uses similar ideas to [7] including the idea of *pointed-specifications* and *singular samplers*[13] with infinite mean target size *cf.* [2]. Constructors are sampled using the inversion method. To compare `boltzmann-samplers` with our prototype framework, we used a rejection-based sampler to generate 100 random λ-terms of sizes in between 800 and 1,200 (using 1,000 as the expected target size).

mean time	9.715 s
standard deviation	671.5 ms

Note that `boltzmann-samplers` is over 92 times slower than an analogous sampler compiled using our framework. A ceiled, rejection-based singular sampler performs a bit better, although it is still over 54 times slower

mean time	5.689 s
standard deviation	224.3 ms

Branching processes. In QuickCheck [8], the prominent framework for random testing in Haskell, users can control the outcome distribution of user-declared generators through, among other things, custom constructor weights influencing the constructor distribution. Unfortunately, it

[11] https://hackage.haskell.org/package/criterion

[12] https://hackage.haskell.org/package/boltzmann-samplers

[13] Let us note that it is possible to approximate them with arbitrary precision using large, finite parameter values.

is quite difficult to *rigorously* control the outcome distribution of so-defined generators. To overcome these challenges, the authors of [22] proposed to adopt *branching processes* to derive QuickCheck generators.

In this approach, the user-declared target outcome distribution is used to compute an apt map of constructor weights leading to QuickCheck generators satisfying the requested constructor distribution. These computations are performed at compile-time and so, similarly to Boltzmann model tuning, there is no additional run-time overhead.

As with other frameworks, we compared our prototype implementation with `DRaGeN` implementing the ideas of [22] using our running example of generating λ-terms of mean size 1,000 and a uniform outcome distribution:

mean time	366.6 ms
standard deviation	32.46 ms

Note that in this benchmark, our prototype is more than 3 times faster.

Let us remark that there is a significant difference in compilation times between `DRaGeN` and our prototype. The branching process computations require time which is proportional to the target size, unlike Boltzmann model tuning which depends on the *bit representation length* of this value. Consequently, Boltzmann samplers can be compiled much quicker allowing users to derive samplers for significantly larger mean parameter values.

Enumeration generators. If uniform outcome distribution is required, one can resort to *enumerative* random generators which injectively encode the inhabitants of a target algebraic data type to consecutive natural numbers. In addition, such maps are size-monotonic and so inhabitants of equal size correspond to a range of natural numbers $[n_1, n_2]$. It is therefore possible to leverage a natural number generator to uniformly sample from $[n_1, n_2]$ and *decode* a corresponding inhabitant by inverting the encoding map.

The `feat` [13] library is one prominent example of such a sampling scheme in Haskell. It supports the enumeration of algebraic data types, and (uniform) random generation of their inhabitants. Generating 100 random λ-terms of (exact) size 1,000 has the following performance

mean time	174.1 ms
standard deviation	16.45 ms

Note that `feat` is more that 1.6 times slower than Boltzmann sampler with mean size 1,000 and a 10% size tolerance. Our sampler outperforms the `feat` one even in the case of a 1% size tolerance. In the *exact-size* sampling regime, however, implemented Boltzmann samplers are no longer linear, but have a quadratic $O(n^2)$ average runtime complexity. Consequently, for small or moderate sizes where enumeration generators are feasible `feat` becomes more efficient.

Let us remark, however, that recent theoretical improvements to Boltzmann samplers in the exact-size regime have brought their $O(n^2)$ average complexity down to $O(n)$, *cf.* [24, 28].

8 Conclusions

We presented a novel framework for the automatic derivation of multi-parametric Boltzmann samplers. With a clean separation of concerns, we provided a declarative and highly modular prototype Haskell implementation which matches, or vastly outperforms several prominent random generation frameworks in a moderate to large outcome size regime.

Given a set of user-declared size notion, target constructor frequencies and size, our framework synthesises efficient, entropy-optimal Boltzmann samplers. Suitable branching probabilities are obtained at compile time through a series of conceptually simpler, intermediate steps. First, the tuning problem is expressed in a specialised and type safe eDSL called `paganini-hs`. The eDSL composes an optimisation problem in another, python-based DSL called `paganini`. There, the domain optimisation problem is further broken down into a convex optimisation problem expressed in yet another DSL called `CVXPY`. The `CVXPY` framework chooses a suitable solver, finds an apt starting point, and solves the convex optimisation problem. Its result is then hoisted through the series of eDSL back into to our framework. Let us notice that each of these intermediate steps forms a separate conceptual module in our framework, each having a clean, distinct set of responsibilities. In particular, the tuning engine forms a separate module which can be used for other purposes or in other frameworks.

While our framework is not unduly optimised, it already exhibits the practical potential of Boltzmann samplers in the field of random generation, so prominently used throughout the functional programming language community. Our benchmarks suggest that Boltzmann samplers are an effective tool in generating large, random inhabitants of algebraic data types. With the additional feature of parameter tuning, it is possible to control not only the size of generated objects, but also their expected shape and form, such as the constructor frequencies. Consequently, multi-parametric Boltzmann samplers form a versatile random generation platform combining rigorous control over the outcome distribution with a convenient, declarative user interface.

Acknowledgments

We would like to express our thanks to Olivier Bodini, Matthieu Dien, Sergey Dovgal, Agustín Mista, Pierre Lescanne, Martin Pépin, and Li-yao Xia for our encouraging discussions.

References

[1] Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. 2021. Tuning as convex optimisation: a polynomial tuner for multi-parametric combinatorial samplers. *Combinatorics, Probability and Computing* (2021), 1–47. https://doi.org/10.1017/S0963548321000547

[2] Olivier Bodini, Antoine Genitrini, and Nicolas Rolin. 2015. Pointed versus singular Boltzmann samplers: a comparative analysis. *Pure Mathematics and Application* 25, 2 (2015), 115–131.

[3] Olivier Bodini, Jérémie Lumbroso, and Nicolas Rolin. 2015. Analytic samplers and the combinatorial rejection method. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*. 40–50.

[4] Olivier Bodini and Yann Ponty. 2010. Multi-dimensional Boltzmann sampling of context-free languages. In *21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA'10)*, Vol. AM.

[5] Olivier Bodini, Olivier Roussel, and Michèle Soria. 2012. Boltzmann samplers for first-order differential specifications. *Discrete Applied Mathematics* 160, 18 (2012), 2563–2572. https://doi.org/10.1016/j.dam.2012.05.022 V Latin American Algorithms, Graphs, and Optimization Symposium — Gramado, Brazil, 2009.

[6] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. Safe Zero-Cost Coercions for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 189–202. https://doi.org/10.1145/2628136.2628141

[7] Benjamin Canou and Alexis Darrasse. 2009. Fast and Sound Random Generation for Automated Testing and Benchmarking in Objective Caml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML* (Edinburgh, Scotland) *(ML '09)*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/1596627.1596637

[8] 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*. 268–279.

[9] Nicolaas G. de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392.

[10] Luc Devroye. 1986. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, USA.

[11] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5.

[12] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. 2004. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing* 13, 4-5 (2004), 577–625.

[13] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. *SIGPLAN Not.* 47, 12 (sep 2012), 61–72. https://doi.org/10.1145/2430532.2364515

[14] Philippe Flajolet, Éric Fusy, and Carine Pivoteau. 2007. Boltzmann sampling of unlabelled structures. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*. 201–211.

[15] Philippe Flajolet, Maryse Pelletier, and Michèle Soria. 2011. On Buffon Machines and Numbers. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, Dana Randall (Ed.). SIAM, 172–183. https://doi.org/10.1137/1.9781611973082.15

[16] Philippe Flajolet and Robert Sedgewick. 2009. *Analytic Combinatorics*. Cambridge University Press.

[17] Michael Grant, Stephen Boyd, and Yinyu Ye. 2006. *Disciplined Convex Programming*. Springer US, Boston, MA, 155–210. https://doi.org/10.1007/0-387-30528-9_7

[18] Katarzyna Grygiel and Pierre Lescanne. 2015. Counting and generating terms in the binary lambda calculus. *Journal of Functional Programming* 25 (2015), e24. https://doi.org/10.1017/S0956796815000271

[19] D. Knuth and A. Yao. 1976. *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, Chapter The complexity of nonuniform random number generation.

[20] Jérémie O. Lumbroso. 2013. Optimal Discrete Uniform Generation from Coin Flips, and Applications. *CoRR* abs/1304.1916 (2013). arXiv:1304.1916 http://arxiv.org/abs/1304.1916

[21] Agustín Mista and Alejandro Russo. 2020. BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 25–46.

[22] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching Processes for QuickCheck Generators. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3242744.3242747

[23] Yurii Nesterov and Arkadii Nemirovskii. 1994. *Interior-Point Polynomial Algorithms in Convex Programming*. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611970791 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611970791

[24] Konstantinos Panagiotou, Leon Ramzews, and Benedikt Stufler. 2021. Exact-size Sampling of Enriched Trees in Linear Time. https://doi.org/10.48550/ARXIV.2110.11472

[25] Carine Pivoteau, Bruno Salvy, and Michèle Soria. 2012. Algorithms for combinatorial structures: Well-founded systems and Newton iterations. *Journal of Combinatorial Theory, Series A* 119, 8 (2012), 1711–1773. https://doi.org/10.1016/j.jcta.2012.05.007

[26] Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Optimal Approximate Sampling from Discrete Probability Distributions. *Proc. ACM Program. Lang.* 4, POPL, Article 36 (dec 2019), 31 pages. https://doi.org/10.1145/3371104

[27] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. *SIGPLAN Not.* 37, 12 (dec 2002), 60–75. https://doi.org/10.1145/636517.636528

[28] Andrea Sportiello. 2021. Boltzmann sampling of irreducible context-free structures in linear time. https://doi.org/10.48550/ARXIV.2105.12881

[29] John Tromp. 2006. Binary Lambda Calculus and Combinatory Logic. In *Kolmogorov Complexity and Applications (Dagstuhl Seminar Proceedings (DagSemProc), Vol. 6051)*, Marcus Hutter, Wolfgang Merkle, and Paul M.B. Vitanyi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–20. https://doi.org/10.4230/DagSemProc.06051.4

[30] Herbert S. Wilf. 2006. *Generatingfunctionology*. A. K. Peters, Ltd., USA.