Effective random generation of algebraic data types through Boltzmann samplers

Maciej Bendkowski

May 18th, 2023

Motivating example: General purpose codecs

Suppose that we want to develop a general purpose codec:

```haskell
class Codec a where
    encode :: a -> ByteString
    decode :: ByteString -> a
```

such that

- `encode . decode = id @a`
- `decode . encode = id @ByteString`
- `size (encode a) ≤ size a`

where

```haskell
class Sized a where
    size :: a -> Int
```

Motivating example: General purpose codecs

Idea

Let's implement a *dictionary based* encoder for `String`, in the spirit of Lempel-Ziv-78, and then figure out how to encode data types as `String`.

Motivating example: General purpose codecs

Encoding:

- initiate an alphabet $\Sigma = \{a, b, c, ..\}$
- scan the input $w_1 \dots w_n$ for its *longest prefix* $w_1 \dots w_p \in \Sigma$
- insert $w_1 \dots w_p w_{p+1}$ into Σ and output the position of $w_1 \dots w_p$ in Σ
- repeat until the whole input is encoded

```
1 | encode :: String -> [Int]
```

Example:

Σ	Input	Output
$\Sigma = \{a, b\}$	abaaa	0
$\Sigma = \{a, b, ab\}$	baaa	0 1
$\Sigma = \{a, b, ab, ba\}$	aaa	0 1 0
$\Sigma = \{a, b, ab, ba, aa\}$	aa	0 1 0 4

Motivating example: General purpose codecs

Decoding:

- initiate an alphabet $\Sigma = \{a, b, c, ..\}$
- process pairs of successive code words (c_i, c_{i+1}).
- for each pair, insert $c_i x$ into Σ where x is the initial letter of c_{i+1}
- output the c_ith symbol w_{c_i} in Σ
- repeat until the whole input is decoded

```
1 | decode :: [Int] -> String
```

Example:

Σ	Input	Output
$\Sigma = \{a, b\}$	0 1 0 4	a
$\Sigma = \{a, b, ab\}$	1 0 4	a b
$\Sigma = \{a, b, ab, ba\}$	0 4	a b a
$\Sigma = \{a, b, ab, ba, aa\}$	4	a b a aa

Motivating example: General purpose codecs

How to en- and decode `BinTree`s?

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

-- | >>> code (Node Leaf (Node Leaf Leaf)) == "10100"
code :: BinTree -> String
code = \case
    Leaf -> "0"
    Node l r -> "1" <> code l <> code r

uncode :: String -> BinTree
uncode = (...)
```

Motivating example: General purpose codecs

```haskell
instance Codec BinTree where
    encode :: BinTree -> ByteString
    encode = Binary.encode . LZ.encode . code

    decode :: ByteString -> BinTree
    decode = uncode . LZ.decode . Binary.decode

instance Size BinTree where
    size :: BinTree -> Int
    size = \case
        Leaf -> 1
        Node lt rt -> 1 + size lt + size rt
```

Motivating example: General purpose codecs

In order to test our implementation, we resort to `QuickCheck` and generate random `BinTree`s to test the specified invariants:

```
1  newtype ArbBinTree = ArbBinTree BinTree
2  instance Arbitrary ArbBinTree where
3      arbitrary = ArbBinTree <$> genBinTree
4    where
5      genBinTree =
6          frequency
7              [ (1, pure Leaf)
8              , (1, Node <$> genBinTree <*> genBinTree)
9              ]
```

Motivating example: General purpose codecs

```
1  testProperty "Codec can de- and encode" $
2      \(ArbBinTree bt) ->
3          Codec.decode (Codec.encode bt) == bt
4
5  testProperty "Encoded trees are compressed" $
6      \(ArbBinTree bt) ->
7          size (encode s) <= size s
```

Our implementation works, but *how good is it*?

Motivating example: General purpose codecs

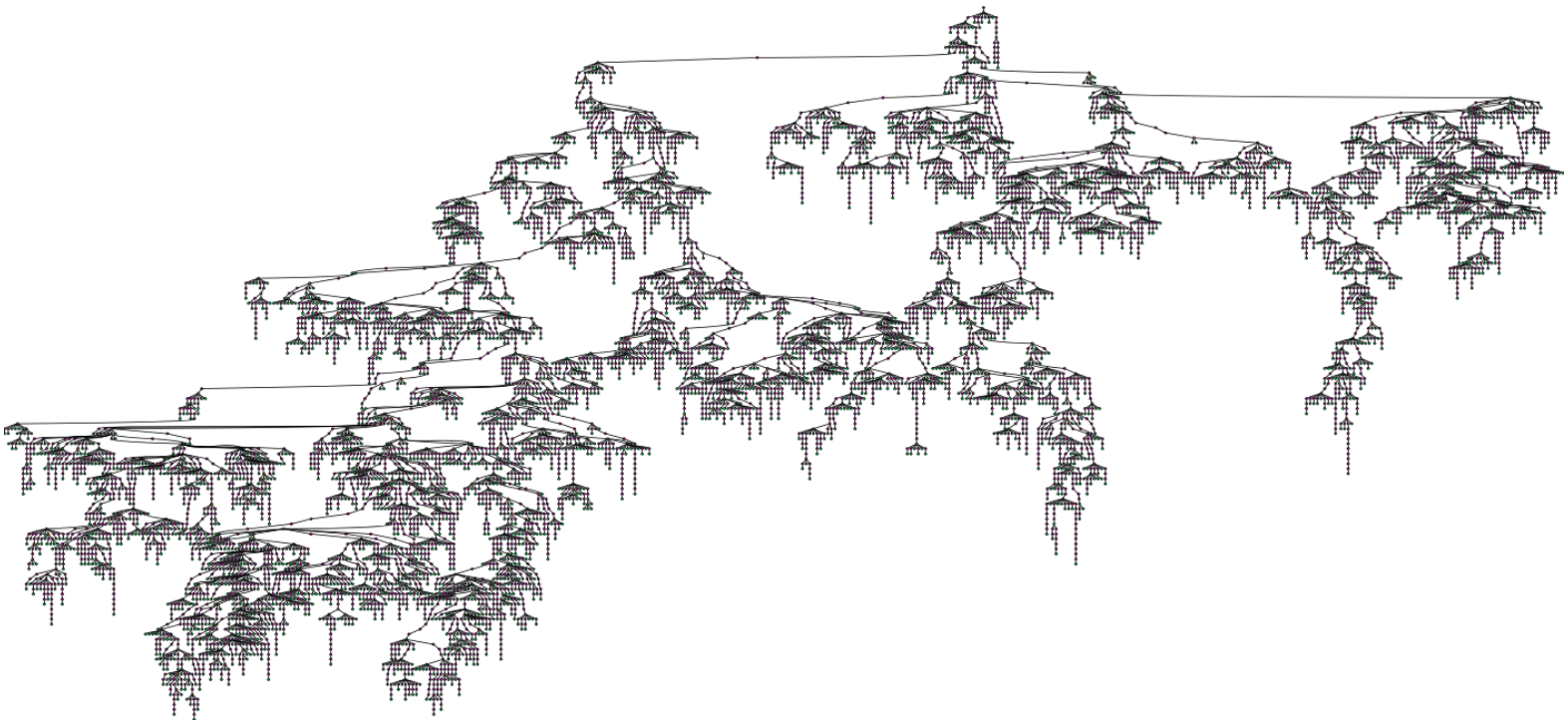
In order to *evaluate* the performance of our prototype codec we can take the average compression ratio over a large corpus of *random* binary trees.

Problem

- What `BinTree` distribution should we choose?
- How to generate `BinTree`s following the chosen distribution?
- How to deal with more involved algebraic data types?

Example: Random λ-terms

OK, let's write a generator for λ-terms in the DeBruijn notation!



Example: Random λ-terms

```haskell
data DeBruijn
  = Z | S DeBruijn

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

instance Arbitrary DeBruijn where
    arbitrary =
        frequency
            [ (?, pure Z)
            , (?, S <$> arbitrary)
            ]

instance Arbitrary Lambda where
    arbitrary =
        frequency
            [ (?, Index <$> arbitrary)
            , (?, Abs <$> arbitrary)
            , (?, App <$> arbitrary <*> arbitrary)
            ]
```

Example: Random λ-terms

```haskell
data DeBruijn
  = Z | S DeBruijn

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

instance Arbitrary DeBruijn where
    arbitrary =
        frequency
            [ (0.509308127, pure Z)
            , (0.49069187299999995, S <$> arbitrary)
            ]

instance Arbitrary Lambda where
    arbitrary =
        frequency
            [ (0.3703026, Index <$> arbitrary)
            , (0.25939476, Abs <$> arbitrary)
            , (0.3703026, App <$> arbitrary <*> arbitrary)
            ]
```

Boltzmann samplers

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

mkDefBoltzmannSampler ''Lambda 10_000
```

Key properties:

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

Source code: See

`https://github.com/maciej-bendkowski/generic-boltzmann-brain`

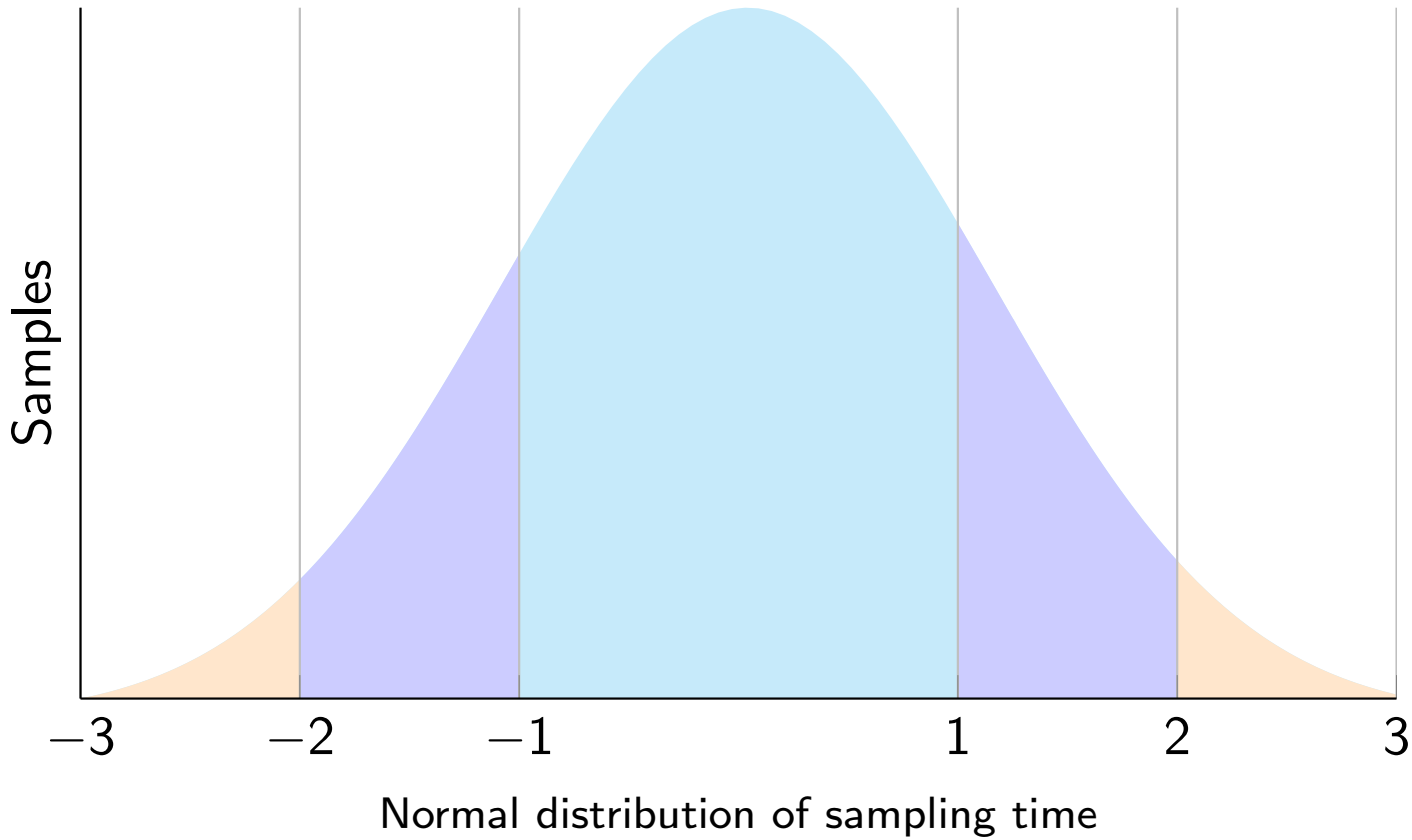
Boltzmann samplers

With `BoltzmannSampler a` we can have a finer control over the size of outcome samples, e.g.:

```
1  rejectionSampler ::
2    (RandomGen g, BoltzmannSampler a) =>
3      LowerBound -> UpperBound -> BitOracle g a
4  rejectionSampler lb ub = do
5    runMaybeT (sample ub)
6      >>= ( \case
7              Just (obj, s) ->
8                if lb <= s && s <= ub
9                  then pure obj
10                 else rejectionSampler lb ub
11             Nothing -> rejectionSampler lb ub
12         )
13
```

Sampling efficiency

$$\mathcal{N}(\mu = 1.127\text{s}, \sigma^2 = (40.73\text{ms})^2)$$



Sampling time for a default Boltzmann sampler
generating $1,000$ random λ-terms of size in between 800 and $1,200$.

Beyond uniform outcome distribution

We don't have to use the *uniform* outcome distribution!

```
1  mkBoltzmannSampler
2    System
3    { targetType = ''Lambda
4    , meanSize = 10_000
5    , frequencies =
6        ('Abs, 4_000) <:> def
7    , weights =
8        ('Index, 0)
9        <:> $(mkDefWeights ''Lambda)
10   }
```

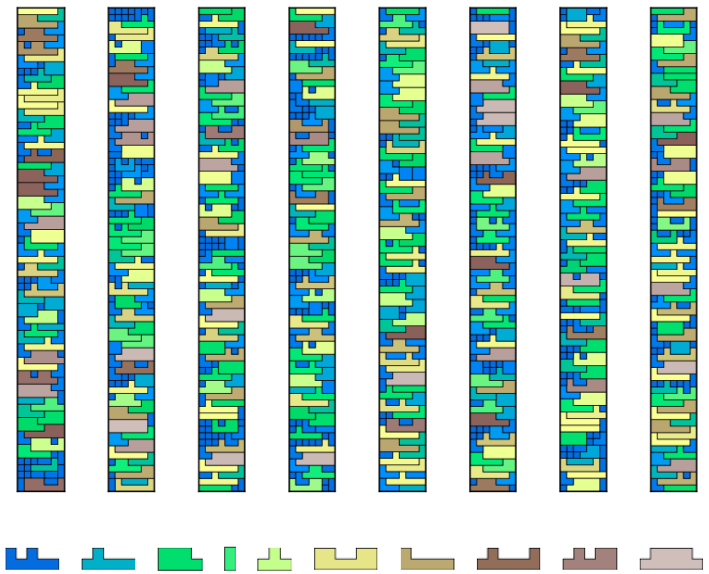
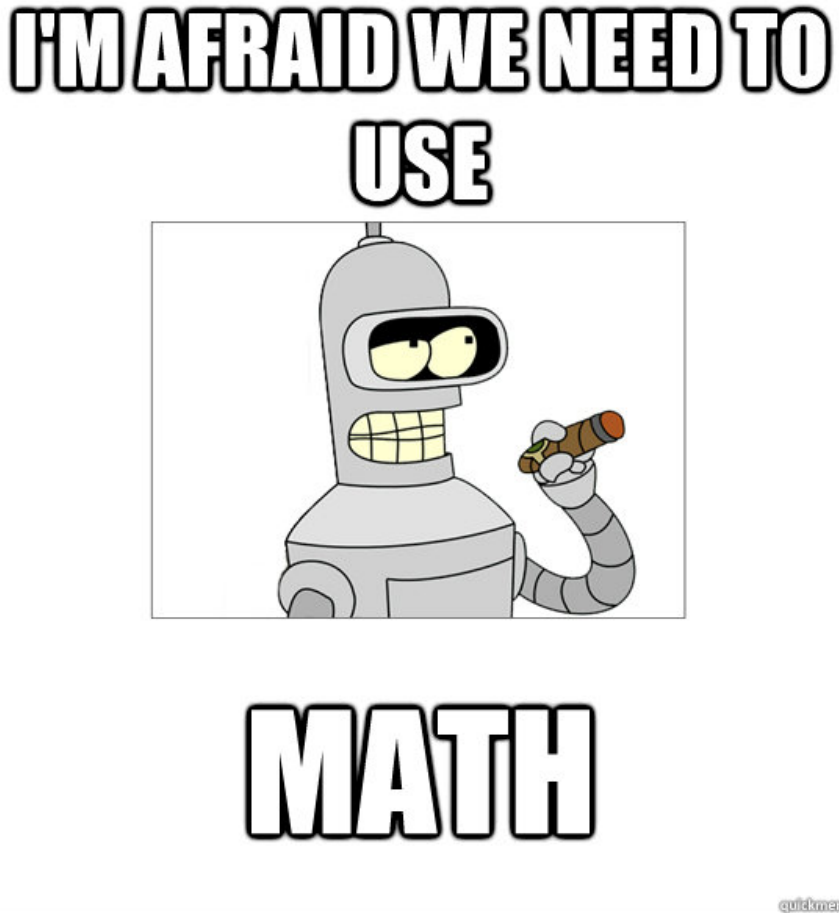


Figure: Five random $n \times 7$ tillings of areas in the interval $[500; 520]$ using in total 95 different tiles.

How do Boltzmann samplers work?



Univariate Boltzmann models

Let $\mathcal{S}$ be a set of objects endowed with a *size* function $|\cdot|: \mathcal{S} \to \mathbb{N}$ with the property that for all $n \geq 0$ the set of objects of size n in $\mathcal{S}$ is *finite*. For such a class of objects, the corresponding *generating function* $S(z)$ is a formal power series

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

whose coefficients $(s_n)_{n \geq 0}$ denote the number of objects of size n in $\mathcal{S}$.

Example

Fibonacci numbers have a generating function $F(z)$ of the form

$$F(z) = 1 + z + 2z^2 + 3z^3 + 5z^4 + 8z^5 + 13z^6 + 21z^7 + \cdots$$

Univariate Boltzmann models

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

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

Note that

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

The probability $\mathbb{P}_x(N = n)$ that the outcome size is n satisfies

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

whereas

$$\mathbb{E}_x(N) = x\frac{\frac{\partial}{\partial x}S(x)}{S(x)} \qquad \sigma_x(N) = \sqrt{x\frac{\frac{\partial}{\partial x}S(x)}{S(x)}}$$

Univariate Boltzmann models

Boltzmann models admit elegant closure properties with respect to algebraic data type constructors, i.e. products and co-products (sums).

$$\{\omega\} \longrightarrow z^{|\omega|}$$

$$\mathcal{A} + \mathcal{B} \longrightarrow A(z) + B(z) = \sum_{n \geq 0} (a_n + b_n) z^n$$

$$\mathcal{A} \times \mathcal{B} \longrightarrow A(z) \times B(z) = \sum_{n \geq 0} \left(\sum_{k=0}^{n} a_k \, b_{n-k} \right) z^n$$

Example

The generating function $F(z)$ for Fibonacci numbers satisfies

$$F(z) = z + z^2 + \left(zF(z) + z^2 F(z) \right) = \frac{z}{1 - z - z^2}$$

Univariate Boltzmann models

Given an algebraic data type $\mathcal{S}$ and a target *mean size* n we can find the *branching probabilities* by solving

$$x\frac{\frac{\partial}{\partial x}S(x)}{S(x)} = n$$

for the (unknown) x.

Problem

How can we solve this equation automatically, in the general case, of systems of possibly mutually recursive algebraic data types?

Convex optimisation: Random λ-terms

```haskell
data DeBruijn
  = Z
  | S DeBruijn

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

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

spec.add(l, d + z * u * l + z * l**2)
spec.run_tuner(l)
```

$$\delta \geq \log\left(e^{\zeta} + e^{\zeta+\sigma}\right)$$
$$\lambda \geq \log\left(e^{\delta} + e^{\zeta+\lambda} + e^{\zeta+2\lambda}\right)$$
$$\lambda - 10,000\zeta \to \min_{\lambda,\sigma,\zeta}$$

Architecture

We separate our concerns through a stack of (e)DSLs:

- `generic-boltzmann-brain` uses `paganini-hs`[1] to formulate a *tuning problem* for `paganini`, and then synthesises the sampler code at compile-time once branching probabilities are computed.
- `paganini`[2], a Python library, translates the domain-specific tuning problem into an optimisation problem using `cvxpy`, a modelling language for convex optimisation problems.
- `cvxpy`[3] uses `cvx` and a variety of optimisation solvers to find a solution to the original tuning problem.

[1] `https://github.com/maciej-bendkowski/paganini-hs`
[2] `https://github.com/maciej-bendkowski/paganini`
[3] `https://www.cvxpy.org/`

Remarks

- `generic-boltzmann-brain` supports `newtype`s, and can be used if multiple samplers or size notions are simultaneously required,
- alas *polymorphic data types*, such as

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

are *not* supported. Since `a` is unknown at compile time, we cannot infer the *concrete* structure of `BinTree a` and formulate the corresponding tuning problem.

Conclusions

- `generic-boltzmann-brain`, and Boltzmann samplers in general, are a useful framework to generate large, random structures with additional control over their size and expected shape,
- while the presented framework is written in Haskell, only the code generation part is Haskell-specific. Most of the presented stack is readily applicable to other languages,
- `generic-boltzmann-brain` is really fast and easy to use.

References

- Duchon, Flajolet, Louchard, Schaeffer (2004) Boltzmann Samplers for the Random Generation of Combinatorial Structures
- B. (2022) Automatic compile-time synthesis of entropy-optimal Boltzmann samplers
- B., Bodini, Dovgal (2018) Polynomial tuning of multiparametric combinatorial samplers
- B. (2022) `generic-boltzmann-brain` @ github