Effective random generation of algebraic data types through Boltzmann samplers

Maciej Bendkowski

April 4, 2024

Motivating example: General purpose codecs

Suppose that we want to develop a general purpose codec:

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

such that

- `encode . decode = id`
- `decode . encode = id`
- `size (encode a)` $\leq$ `size a`

where

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

Motivating example: General purpose codecs

Q: How to en- and decode BinTrees?

```
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

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

    decode :: ByteString -> BinTree
    decode = uncode . LempelZiv.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 BinTrees to test the specified invariants:

```
newtype ArbBinTree = ArbBinTree BinTree
instance Arbitrary ArbBinTree where
    arbitrary = ArbBinTree <$> genBinTree
  where
    genBinTree =
        frequency
            [ (1, pure Leaf)
            , (1, Node <$> genBinTree <*> genBinTree)
            ]
```

Motivating example: General purpose codecs

```haskell
testProperty "Codec can de- and encode" $
    \(ArbBinTree bt) ->
        Codec.decode (Codec.encode bt) == bt

testProperty "Encoded trees are compressed" $
    \(ArbBinTree bt) ->
        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

Example: Random λ-terms

```
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

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

mkDefBoltzmannSampler ''Lambda 10_000
```

Key properties:

- the outcome *size* of `sample` is a random variables,
- 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`.

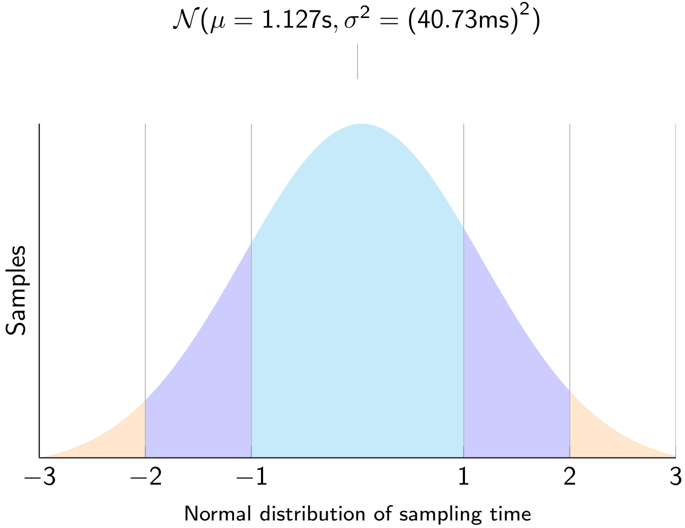
Boltzmann samplers

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

```
rejectionSampler ::
  (RandomGen g, BoltzmannSampler a) =>
    LowerBound -> UpperBound -> BitOracle g a
rejectionSampler lb ub = do
  runMaybeT (sample ub)
    >>= ( \case
            Just (obj, s) ->
              if lb <= s && s <= ub
                then pure obj
                else rejectionSampler lb ub
            Nothing -> rejectionSampler lb ub
        )

```

Benchmarks



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!

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

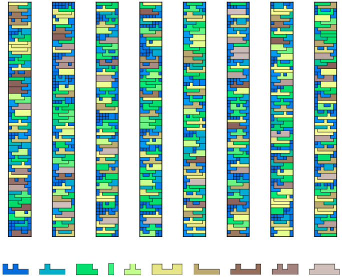


Figure: Five random $n \times 7$ tillings of areas in the interval $[500; 520]$ using in total 95 different tiles.

Generating functions

Let $\mathcal{S}$ be a set of objects endowed with a *size* function $|\cdot|\colon \mathcal{S} \to \mathbb{N}$ 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

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

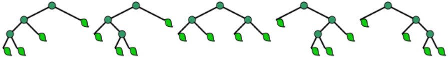
whose coefficients $(s_n)_{n \geq 0}$ denote the number of objects of size n in $\mathcal{S}$.

Example

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

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

Generating functions



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

The recursive *sum-of-products* nature of algebraic data types lets us find the functional recursion defining the respective generating function:

$$\begin{aligned} C(z) &= 1 + z + 2z^2 + 5z^3 + 14z^4 + 42z^5 + \cdots \\ &= 1 + zC(z)^2 \\ &= \frac{1 - \sqrt{1 - 4z}}{2z} \end{aligned}$$

Remark

Usually, we cannot find closed-form solutions for generating functions.

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 a 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

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

Parameter tuning

Problem

Given a system of (possibly mutually recursive) algebraic data types, and a target outcome size n, find the respective control parameter x such that

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

Parameter tuning

Problem

Given a system of (possibly mutually recursive) algebraic data types, and a target outcome size n, find the respective control parameter x such that

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

Solution (not covered in this presentation)

Transform the system of partial differential equation into an equivalent *convex optimisation* problem, solve the optimisation problem using efficient interior-point methods, and translate the solution back to the original domain.

Composition rules for sampler construction

Q: Suppose that we have the control parameter x.
How to turn it into an efficient sampler?

- Let $\mathcal{S} = \mathcal{A} + \mathcal{B}$ be a co-product class. Then, for $\omega \in \mathcal{S}$

$$\mathbb{P}_x(\omega \in \mathcal{A}) = \frac{\sum_{\gamma \in \mathcal{A}} z^{|\gamma|}}{S(x)} = \frac{A(x)}{S(x)}$$

  Hence, before sampling an object from $\mathcal{S}$, we need to make a *skewed coin toss*. With probability $\frac{A(x)}{S(x)}$ we invoke the sampler for $\mathcal{A}$. With probability $\frac{B(x)}{S(x)}$ we invoke the sampler for $\mathcal{B}$.

- Let $\mathcal{S} = \mathcal{A} \times \mathcal{B}$ be a product class. Then, for $\omega = (\alpha, \beta)$

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

  Hence, we can invoke both samplers for $\mathcal{A}$ and $\mathcal{B}$ and pair up their results.

Divide and conquer

Compiling Boltzmann samplers breaks down into the following sub-problems:

- collect the system of ADTs and user parameters (target size, etc.),
- find all the necessary tuning parameters,
- compute *branching probabilities* for co-product types,
- generate *efficient* Haskell code for the sampler.

Code generation

```
instance BoltzmannSampler Lambda where
 {-# INLINE sample #-}
  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)
```

Sampling from a finite distribution: some ideas

Q: Suppose that we have computed *branching probabilities* $\mathbb{P} = (p_1, \ldots, p_n)$ for a data type with n distinct constructors. How can we effectively generate from $\mathbb{P}$?

Sampling from a finite distribution: some ideas

Q: Suppose that we have computed *branching probabilities* $\mathbb{P} = (p_1, \ldots, p_n)$ for a data type with n distinct constructors. How can we effectively generate from $\mathbb{P}$?

Solution

Draw a random number $x \in [0, 1]$ using a standard library call, find the segment $[p_i, p_{i+1})$ containing x, and use the respective constructor.

Sampling from a finite distribution: some ideas

Q: Suppose that we have computed *branching probabilities* $\mathbb{P} = (p_1, \ldots, p_n)$ for a data type with n distinct constructors. How can we effectively generate from $\mathbb{P}$?

Solution

Draw a random number $x \in [0, 1]$ using a standard library call, find the segment $[p_i, p_{i+1})$ containing x, and use the respective constructor.

Better solution

Note that to draw x we need about 32 random bits whereas we might need much less, e.g. for $\mathbb{P} = (0.5, 0.5)$. We can therefore bisect the probability space using one random bit at a time.

Sampling from a finite distribution: some ideas

Q: Suppose that we have computed *branching probabilities* $\mathbb{P} = (p_1, \ldots, p_n)$ for a data type with n distinct constructors. How can we effectively generate from $\mathbb{P}$?

Solution

Draw a random number $x \in [0, 1]$ using a standard library call, find the segment $[p_i, p_{i+1})$ containing x, and use the respective constructor.

Better solution

Note that to draw x we need about 32 random bits whereas we might need much less, e.g. for $\mathbb{P} = (0.5, 0.5)$. We can therefore bisect the probability space using one random bit at a time.

Even Better solution

Use the idea of *discrete distribution trees* devised by Knuth and Yao.

Multiple samplers at a time

```
data DeBruijn
  = Z
  | S DeBruijn

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

newtype BinLambda = MkBinLambda Lambda
mkBoltzmannSampler ''BinLambda 1_000

-- compiler-generated type
newtype Gen_DeBruijn = MkGen_DeBruijn DeBruijn

-- type coercion(s)
coerce
  @(DeBruijn -> BinLambda)
  @(Gen_DeBruijn -> BinLambda) Index
```

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)
```

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

References

Boltzmann samplers compiler:
https://github.com/maciej-bendkowski/generic-boltzmann-brain