Effective random generation of algebraic data types through Boltzmann samplers

Maciej Bendkowski

May 18th, 2023

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)` $\leq$ `size a`

where

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

Example: General purpose codecs

Let's start with implementing a *dictionary based* encoder for `String`, in the spirit of Lempel-Ziv-78.

Example: General purpose codecs

Encoding:

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

```
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

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

```haskell
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

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 = (...)
```

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
```

Example: General purpose codecs

In order to test our implementation, we resort to `QuickCheck` and generate random `BinTree`s 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)
            ]
```

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*?

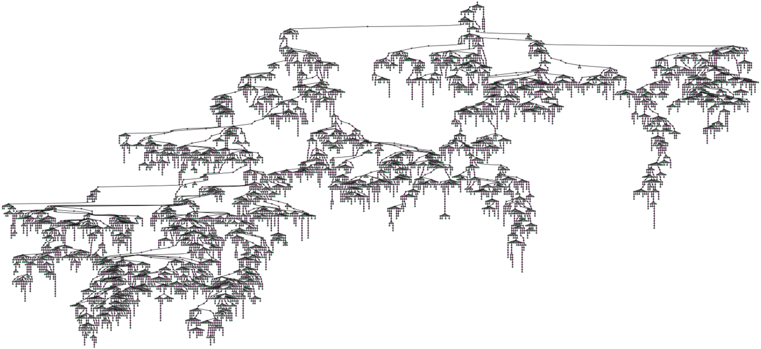
Example: General purpose codecs

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.

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

Example: Random λ-terms

Let's write a generator for λ-terms in the DeBruijn notation!



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

```
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
```

- 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`.

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

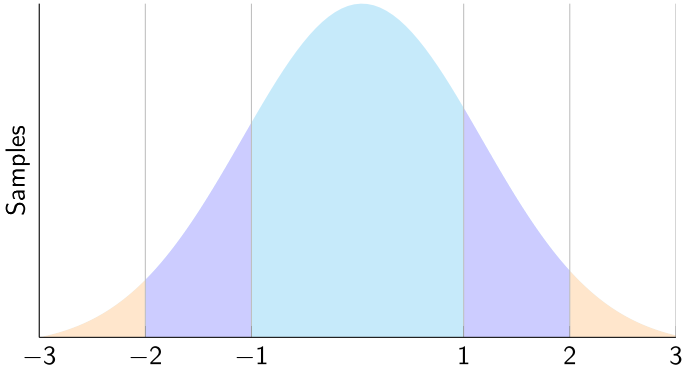
Boltzmann samplers

```
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
          )

```

Sampling efficiency

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



Normal distribution of sampling time

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

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

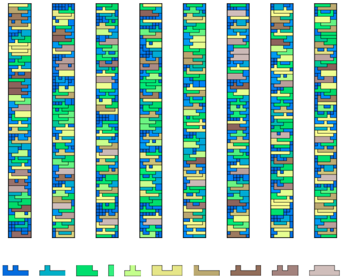


Figure: Five random $n \times 7$ tillings of areas in the interval $[500; 520]$ using in total 95 different tiles.

Univariate Boltzmann models

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

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

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).

$$
\begin{aligned}
\{\omega\} &\longrightarrow z^{|\omega|} \\
\mathcal{A} + \mathcal{B} &\longrightarrow A(z) + B(z) = \sum_{n \geq 0} (a_n + b_n) z^n \\
\mathcal{A} \times \mathcal{B} &\longrightarrow A(z) \times B(z) = \sum_{n \geq 0} \left(\sum_{k=0}^{n} a_k b_{n-k} \right) z^n
\end{aligned}
$$

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

```
data DeBruijn
  = Z
  | S DeBruijn

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

$$D(z) = z + zD(z)$$
$$L(z) = D(z) + zL(z) + zL(z)^2$$

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

Architecture

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

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

Remarks

- `newtype`s are supported, and can be used if multiple samplers or size notions are simultaneously required.
- Polymorphic data types, such as

```
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

Thank you for your attention!

References:

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