Enum instances on steroids

Andrew Lelechenko
1@dxdy.ru

Barclays, London

Berlin Functional Programming Group, 21.07.2020

List generators in Haskell

- Generate a finite range:

$$[1 \,..\, 5] = [1, 2, 3, 4, 5].$$

- Generate an infinite range:

$$[1 \,..\,] \;\; = [1, 2, 3, 4, 5 \ldots].$$

- Generate ranges with a step:

$$\begin{aligned} [1, 3 \,..\, 9] &= [1, 3, 5, 7, 9], \\ [1, 3 \,..\,] \;\; &= [1, 3, 5, 7, 9 \ldots]. \end{aligned}$$

Double trouble

- As expected,
$$[0, 1 \,.\,.\, 3] \quad = [0, 1, 2, 3].$$
- But
$$[0, 1 \,.\,.\, 3.5] = [0, 1, 2, 3, 4].$$

Why?

- *Short answer:* because Haskell Report 2010 says so!
- *Longer answer:* floating-point arithmetic is not exact, so without some kind of rounding we'd have equally surprising
$$[0, 0.1 \,.\,.\, 0.3] = [0, 0.1, 0.2],$$
because
$$0.1 + 0.1 + 0.1 = 0.30000000000000004 > 0.3.$$

Not only numbers

- List generators are available for user-defined types:

  $\textit{data}\ \mathrm{Day} = \mathrm{Mon} \mid \mathrm{Tue} \mid \mathrm{Wed} \mid \mathrm{Thu} \mid \mathrm{Fri} \mid \mathrm{Sat} \mid \mathrm{Sun}\ \textit{deriving}\ \mathrm{Enum}$

- Generate a range:

$$[\mathrm{Tue} \mathinner{\ldotp\ldotp} \mathrm{Fri}] = [\mathrm{Tue}, \mathrm{Wed}, \mathrm{Thu}, \mathrm{Fri}].$$

- Generate a range with a step:

$$[\mathrm{Mon}, \mathrm{Wed} \mathinner{\ldotp\ldotp} \mathrm{Sun}] = [\mathrm{Mon}, \mathrm{Wed}, \mathrm{Fri}, \mathrm{Sun}].$$

What happens under the hood?

```
class Enum a where
  enumFrom       :: a → [a]
                 -- 1 → [1,2,3,4,5...]
  enumFromThen   :: a → a → [a]
                 -- 1 → 3 → [1,3,5,7,9...]
  enumFromTo     :: a → a → [a]
                 -- 1 → 5 → [1,2,3,4,5]
  enumFromThenTo :: a → a → a → [a]
                 -- 1 → 3 → 9 → [1,3,5,7,9]
```

We need to go deeper

```
class Enum a where
  fromEnum :: a → Int
  toEnum   :: Int → a

  succ :: a → a -- next
  succ x = toEnum (fromEnum x + 1)

  pred :: a → a -- prev
  pred x = toEnum (fromEnum x - 1)

  enumFrom :: a → [a]
  enumFrom x = map toEnum (iterate (+ 1) (fromEnum x))

  enumFromTo :: a → a → [a]
  enumFromTo x y = map toEnum
    (takeWhile (⩽ fromEnum y) (iterate (+ 1) (fromEnum x)))
```

Partial functions

- `succ maxBound` / `pred minBound` are undefined.

- Most of enumerable data is “smaller” than `Int`, so `toEnum` :: `Int` $\to$ a is a partial function.
- Others (`Integer`, `Natural`) are “bigger”, so `fromEnum` :: a $\to$ `Int` has a truncating behavior.
- The only faithful instance is `Enum Int`.

- `fromEnum` and `toEnum` have utility, reaching far beyond list comprehensions:
  - Generating random values.
  - Faster maps and sets, backed by `IntMap` and `IntSet`.
  - Storing values in unboxed vectors.

No built-in deriving

Biggest issue: `Enum` is derivable only for types, whose constructors have no fields. This does not work:

$$\begin{aligned} &\textit{data}\ \mathsf{WeekDay} = \mathsf{Mon} \mid \mathsf{Tue} \mid \mathsf{Wed} \mid \mathsf{Thu} \mid \mathsf{Fri} && \textit{deriving}\ \mathsf{Enum} \\ &\textit{data}\ \mathsf{WeekEnd} = \mathsf{Sat} \mid \mathsf{Sun} && \textit{deriving}\ \mathsf{Enum} \\ &\textit{data}\ \mathsf{Day} = \mathsf{Work}\ \mathsf{WeekDay} \mid \mathsf{Play}\ \mathsf{WeekEnd} && \textit{deriving}\ \color{red}{\mathsf{Enum}} \end{aligned}$$

This also does not work:

$$[\text{Nothing} \ldots \text{Just True}] :: [\text{Maybe Bool}]$$
$$[\text{Left } () \ldots \text{Right True}] :: [\text{Either } () \text{ Bool}]$$

Cardinality

- *Cardinality* is just a fancy word for the number of values inhabiting a type.
  - Cardinality of `Bool` is 2.
  - Cardinality of () is 1.
  - Cardinality of `Void` is 0.
- Cardinality of `data Foo = Foo Bool Bool | Bar ()` equals to $2 \times 2 + 1 = 5$.
- In its essence `Enum` is about finitely-inhabited types, however it lacks means to infer their cardinality, which makes it unreliable and its instances difficult to define.

```
class MyEnum a where
  cardinality :: Proxy a → Integer
  toMyEnum    :: Integer → a
  fromMyEnum  :: a → Integer
```

Products and sums of types

- Each *algebraic* data type is isomorphic to a sum of products of its constituents, which is called its *generic* representation.
- `data Foo = Foo Bool Bool` is isomorphic to `(Bool, Bool)` or in other notation Bool $:\times:$ Bool.
- `data Foo = Bar Bar | Baz Baz` is isomorphic to `Either Bar Baz` or in other notation Bar $:+:$ Baz.
- `data Foo = Foo` is isomorphic to () type.
- `data Foo = Foo Bool Bool | Bar` is isomorphic to `Either (Bool, Bool) ()` or in other notation Bool $:\times:$ Bool $:+:$ ().
- GHC provides an automatic way to convert between types and their generic representations.

Generic instance for sum

```
class GMyEnum f where
  gcardinality :: Proxy f → Integer
  toGMyEnum    :: Integer → f a
  fromGMyEnum  :: f a → Integer

instance (GMyEnum a, GMyEnum b) ⇒ GMyEnum (a :+: b) where
  gcardinality _ =
    gcardinality (Proxy @a) + gcardinality (Proxy @b)

  toGMyEnum n | n < cardA = L1 (toGMyEnum n)
              | otherwise = R1 (toGMyEnum (n - cardA))
    where cardA = gcardinality (Proxy @a)

  fromGMyEnum = \case
     L1 x → fromGMyEnum x
     R1 x → fromGMyEnum x + gcardinality (Proxy @a)
```

Generic instance for product

```
instance (GMyEnum a, GMyEnum b) ⇒ GMyEnum (a :*: b) where
  gcardinality _ =
    gcardinality (Proxy @a) * gcardinality (Proxy @b)

  toGMyEnum n = toGMyEnum q :*: toGMyEnum r
    where
      cardB = gcardinality (Proxy @b)
      (q, r) = n `quotRem` cardB

  fromGMyEnum (q :*: r) =
    gcardinality (Proxy @b) * fromGMyEnum q + fromGMyEnum r
```

Full source code available from https://github.com/Bodigrim/random/blob/generic/src/System/Random/GFinite.hs

Example of autoderiving

```haskell
{-# LANGUAGE DeriveGeneric  #-}
{-# LANGUAGE DeriveAnyClass #-}

import GHC.Generics

data Action = Code Bool | Eat Bool Bool | Sleep ()
  deriving (Show, Generic, MyEnum)

> cardinality (Proxy @Action)
7

> map toMyEnum [0..7-1] :: [Action]
[Code False,Code True,Eat False False,Eat False True,
 Eat True False,Eat True True,Sleep ()]
```

Making illegal states unrepresentable

```
class MyEnum a where
  cardinality :: Proxy a → Integer
  toMyEnum    :: Integer → a
  fromMyEnum  :: a → Integer
```

Define `data Finite` (n :: `Nat`), which is inhabited exactly by n values $[0\,..\,n-1]$, and promote cardinality to the type level:

```
class MyEnum a where
  type Cardinality a :: Nat
  toMyEnum   :: Finite (Cardinality a) → a
  fromMyEnum :: a → Finite (Cardinality a)
```

This approach can be found in `finitary` and `finitary-derive`.

Countable...

- A data type is called countable if it is isomorphic to `Integer`: there exist total functions

```
fromCountable :: a → Integer
toCountable   :: Integer → a
```

- `Either Integer Integer` is still countable: map `Left` to even numbers and `Right` to odd numbers.

```
fromCountable :: Either Integer Integer → Integer
fromCountable (Left n)  = n * 2
fromCountable (Right n) = n * 2 + 1

toCountable :: Integer → Either Integer Integer
toCountable n
  | even n    = Left  ( n       `div` 2)
  | otherwise = Right ((n - 1) `div` 2)
```

. . . and uncountable

- `(Integer, Integer)` is also countable: just interleave bits from both coordinates.

```
fromCountable :: (Integer, Integer) → Integer
fromCountable (0b1111, 0b0000) = 0b10101010

toCountable :: Integer → (Integer, Integer)
toCountable 0b10101010 = (0b1111, 0b0000)
```

- So sums and products of countable data are countable again! **But what is uncountable then?**
- Set of functions `Integer → Bool` is uncountable, isomorphic to the set of real numbers.

Extending MyEnum for countable data

- Define

```
data Cardinality = Finite Integer | Countable

class MyEnum a where
  cardinality :: Proxy a → Cardinality
  toMyEnum    :: Cardinality → a
  fromMyEnum  :: a → Cardinality
```

- This makes infinitely-inhabited types such as `[Bool]` "enumerable".
- Implemented in `cantor-pairing` package.

Thank you!

andrew.lelechenko@gmail.com Bodigrim

github.com/Bodigrim/my-talks